GENE H. GOLUB · CHARLES F. VAN LOAN

MATRIX COMPUTATIONS

THIRD EDITION

Matrix Computations

THIRD EDITION

Gene H. Golub
Department of Computer Science
Stanford University

Charles F. Van Loan
Department of Computer Science
Cornell University

The Johns Hopkins University Press
Baltimore and London

Johns Hopkins Studies in the Mathematical Sciences
in association with the Department of Mathematical Sciences
The Johns Hopkins University

Printed in the United States of America on acid-free paper
05 04 03 02 01 00 99 98 97 5 4 3 2

First edition 1983
Second edition 1989
Third Edition 1996

The Johns Hopkins University Press
2715 North Charles Street
Baltimore, Maryland 21218-4319
The Johns Hopkins Press Ltd., London

Library of Congress Cataloging-in-Publication Data will be found at the end of this book.

A catalog record for this book is available from the British Library.

ISBN 0-8018-5413-X
ISBN 0-8018-5414-8 (pbk.)

DEDICATED TO

ALSTON S. HOUSEHOLDER

AND

JAMES H. WILKINSON

Contents

Preface to the Third Edition xi
Software xiii
Selected References xv

1 Matrix Multiplication Problems 1

1.1 Basic Algorithms and Notation 2
1.2 Exploiting Structure 16
1.3 Block Matrices and Algorithms 24
1.4 Vectorization and Re-Use Issues 34

2 Matrix Analysis 48

2.1 Basic Ideas from Linear Algebra 48
2.2 Vector Norms 52
2.3 Matrix Norms 54
2.4 Finite Precision Matrix Computations 59
2.5 Orthogonality and the SVD 69
2.6 Projections and the CS Decomposition 75
2.7 The Sensitivity of Square Linear Systems 80

3 General Linear Systems 87

3.1 Triangular Systems 88
3.2 The LU Factorization 94
3.3 Roundoff Analysis of Gaussian Elimination 104
3.4 Pivoting 109
3.5 Improving and Estimating Accuracy 123

4 Special Linear Systems 133

4.1 The LDM^T and LDL^T Factorizations 135
4.2 Positive Definite Systems 140
4.3 Banded Systems 152
4.4 Symmetric Indefinite Systems 161
4.5 Block Systems 174
4.6 Vandermonde Systems and the FFT 183
4.7 Toeplitz and Related Systems 193

5 Orthogonalization and Least Squares 206

5.1 Householder and Givens Matrices 208
5.2 The QR Factorization 223
5.3 The Full Rank LS Problem 236
5.4 Other Orthogonal Factorizations 248
5.5 The Rank Deficient LS Problem 256
5.6 Weighting and Iterative Improvement 264
5.7 Square and Underdetermined Systems 270

6 Parallel Matrix Computations 275

6.1 Basic Concepts 276
6.2 Matrix Multiplication 292
6.3 Factorizations 300

7 The Unsymmetric Eigenvalue Problem 308

7.1 Properties and Decompositions 310
7.2 Perturbation Theory 320
7.3 Power Iterations 330
7.4 The Hessenberg and Real Schur Forms 341
7.5 The Practical QR Algorithm 352
7.6 Invariant Subspace Computations 362
7.7 The QZ Method for $Ax = \lambda Bx$ 375

8 The Symmetric Eigenvalue Problem 391

8.1 Properties and Decompositions 393
8.2 Power Iterations 405

8.3 The Symmetric QR Algorithm 414
8.4 Jacobi Methods 426
8.5 Tridiagonal Methods 439
8.6 Computing the SVD 448
8.7 Some Generalized Eigenvalue Problems 461

9 Lanczos Methods 470

9.1 Derivation and Convergence Properties 471
9.2 Practical Lanczos Procedures 479
9.3 Applications to $Ax = b$ and Least Squares 490
9.4 Arnoldi and Unsymmetric Lanczos 499

10 Iterative Methods for Linear Systems 508

10.1 The Standard Iterations 509
10.2 The Conjugate Gradient Method 520
10.3 Preconditioned Conjugate Gradients 532
10.4 Other Krylov Subspace Methods 544

11 Functions of Matrices 555

11.1 Eigenvalue Methods 556
11.2 Approximation Methods 562
11.3 The Matrix Exponential 572

12 Special Topics 579

12.1 Constrained Least Squares 580
12.2 Subset Selection Using the SVD 590
12.3 Total Least Squares 595
12.4 Computing Subspaces with the SVD 601
12.5 Updating Matrix Factorizations 606
12.6 Modified/Structured Eigenproblems 621

Bibliography 637
Index 687

Preface to the Third Edition

The field of matrix computations continues to grow and mature. In the *Third Edition* we have added over 300 new references and 100 new problems. The LINPACK and EISPACK citations have been replaced with appropriate pointers to LAPACK with key codes tabulated at the *beginning* of appropriate chapters.

In the *First Edition* and *Second Edition* we identified a small number of global references: Wilkinson (1965), Forsythe and Moler (1967), Stewart (1973), Hanson and Lawson (1974) and Parlett (1980). These volumes are as important as ever to the research landscape, but there are some magnificent new textbooks and monographs on the scene. See *The Literature* section that follows.

We continue as before with the practice of giving references at the end of each section and a master bibliography at the end of the book.

The earlier editions suffered from a large number of typographical errors and we are obliged to the dozens of readers who have brought these to our attention. Many corrections and clarifications have been made.

Here are some specific highlights of the new edition. Chapter 1 (Matrix Multiplication Problems) and Chapter 6 (Parallel Matrix Computations) have been completely rewritten with less formality. We think that this facilitates the building of intuition for high performance computing and draws a better line between algorithm and implementation on the printed page.

In Chapter 2 (Matrix Analysis) we expanded the treatment of CS decomposition and included a proof. The overview of floating point arithmetic has been brought up to date. In Chapter 4 (Special Linear Systems) we embellished the Toeplitz section with connections to circulant matrices and the fast Fourier transform. A subsection on equilibrium systems has been included in our treatment of indefinite systems.

A more accurate rendition of the modified Gram-Schmidt process is offered in Chapter 5 (Orthogonalization and Least Squares). Chapter 8 (The Symmetric Eigenproblem) has been extensively rewritten and rearranged so as to minimize its dependence upon Chapter 7 (The Unsymmetric Eigenproblem). Indeed, the coupling between these two chapters is now so minimal that it is possible to read either one first.

In Chapter 9 (Lanczos Methods) we have expanded the discussion of

the unsymmetric Lanczos process and the Arnoldi iteration. The "unsymmetric component" of Chapter 10 (Iterative Methods for Linear Systems) has likewise been broadened with a whole new section devoted to various Krylov space methods designed to handle the sparse unsymmetric linear system problem.

In §12.5 (Updating Orthogonal Decompositions) we included a new subsection on ULV updating. Toeplitz matrix eigenproblems and orthogonal matrix eigenproblems are discussed in §12.6.

Both of us look forward to continuing the dialog with our readers. As we said in the Preface to the *Second Edition*, "It has been a pleasure to deal with such an interested and friendly readership."

Many individuals made valuable *Third Edition* suggestions, but Greg Ammar, Mike Heath, Nick Trefethen, and Steve Vavasis deserve special thanks.

Finally, we would like to acknowledge the support of Cindy Robinson at Cornell. A dedicated assistant makes a big difference.

Software

LAPACK

Many of the algorithms in this book are implemented in the software package LAPACK:

E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen (1995). *LAPACK Users' Guide, Release 2.0, 2nd ed.*, SIAM Publications, Philadelphia.

Pointers to some of the more important routines in this package are given at the *beginning* of selected chapters:

Chapter 1.	Level-1, Level-2, Level-3 BLAS
Chapter 3.	General Linear Systems
Chapter 4.	Positive Definite and Band Systems
Chapter 5.	Orthogonalization and Least Squares Problems
Chapter 7.	The Unsymmetric Eigenvalue Problem
Chapter 8.	The Symmetric Eigenvalue Problem

Our LAPACK references are spare in detail but rich enough to "get you started." Thus, when we say that `_TRSV` can be used to solve a triangular system $Ax = b$, we leave it to you to discover through the LAPACK manual that A can be either upper or lower triangular and that the transposed system $A^Tx = b$ can be handled as well. Moreover, the underscore is a placeholder whose mission is to designate type (single, double, complex, etc).

LAPACK stands on the shoulders of two other packages that are milestones in the history of software development. EISPACK was developed in the early 1970s and is dedicated to solving symmetric, unsymmetric, and generalized eigenproblems:

B.T. Smith, J.M. Boyle, Y. Ikebe, V.C. Klema, and C.B. Moler (1970). *Matrix Eigensystem Routines: EISPACK Guide, 2nd ed.*, Lecture Notes in Computer Science, Volume 6, Springer-Verlag, New York.

B.S. Garbow, J.M. Boyle, J.J. Dongarra, and C.B. Moler (1972). *Matrix Eigensystem Routines: EISPACK Guide Extension*, Lecture Notes in Computer Science, Volume 51, Springer-Verlag, New York.

LINPACK was developed in the late 1970s for linear equations and least squares problems:

EISPACK and LINPACK have their roots in sequence of papers that feature Algol implementations of some of the key matrix factorizations. These papers are collected in

J.H. Wilkinson and C. Reinsch, eds. (1971). *Handbook for Automatic Computation, Vol. 2, Linear Algebra*, Springer-Verlag, New York.

NETLIB

A wide range of software including LAPACK, EISPACK, and LINPACK is available electronically via Netlib:

World Wide Web:	`http://www.netlib.org/index.html`
Anonymous ftp:	`ftp://ftp.netlib.org`

Via email, send a one-line message:

```
mail netlib@ornl.gov
send index
```

to get started.

MATLAB®

Complementing LAPACK and defining a very popular matrix computation environment is MATLAB:

MATLAB *User's Guide*, The MathWorks Inc., Natick, Massachusetts.

M. Marcus (1993). *Matrices and* MATLAB*: A Tutorial*, Prentice Hall, Upper Saddle River, NJ.

R. Pratap (1995). *Getting Started with* MATLAB, Saunders College Publishing, Fort Worth, TX.

Many of the problems in *Matrix Computations* are best posed to students as MATLAB problems. We make extensive use of MATLAB notation in the presentation of algorithms.

Selected References

Each section in the book concludes with an annotated list of references. A master bibliography is given at the end of the text.

Useful books that collectively cover the field, are cited below. Chapter titles are included if appropriate but do not infer too much from the level of detail because one author's chapter may be another's subsection. The citations are classified as follows:

Pre-1970 Classics. Early volumes that set the stage.
Introductory (General). Suitable for the undergraduate classroom.
Advanced (General). Best for practitioners and graduate students.
Analytical. For the supporting mathematics.
Linear Equation Problems. $Ax = b$.
Linear Fitting Problems. $Ax \approx b$.
Eigenvalue Problems. $Ax = \lambda x$.
High Performance. Parallel/vector issues.
Edited Volumes. Useful, thematic collections.

Within each group the entries are specified in chronological order.

Pre-1970 Classics

V.N. Faddeeva (1959). *Computational Methods of Linear Algebra*, Dover, New York.

Basic Material from Linear Algebra. Systems of Linear Equations. The Proper Numbers and Proper Vectors of a Matrix.

E. Bodewig (1959). *Matrix Calculus*, North Holland, Amsterdam.

Matrix Calculus. Direct Methods for Linear Equations. Indirect Methods for Linear Equations. Inversion of Matrices. Geodetic Matrices. Eigenproblems.

R.S. Varga (1962). *Matrix Iterative Analysis*, Prentice-Hall, Englewood Cliffs, NJ.

Matrix Properties and Concepts. Nonnegative Matrices. Basic Iterative Methods and Comparison Theorems. Successive Overrelaxation Iterative Methods. Semi-Iterative Methods. Derivation and Solution of Elliptic Difference Equations. Alternating Direction Implicit Iterative Methods. Matrix Methods for Parabolic Partial Differential Equations. Estimation of Acceleration Parameters.

J.H. Wilkinson (1963). *Rounding Errors in Algebraic Processes*, Prentice-Hall, Englewood Cliffs, NJ.

The Fundamental Arithmetic Operations. Computations Involving Polynomials. Matrix Computations.

A.S. Householder (1964). *Theory of Matrices in Numerical Analysis*, Blaisdell, New York. Reprinted in 1974 by Dover, New York.

Some Basic Identities and Inequalities. Norms, Bounds, and Convergence. Localization Theorems and Other Inequalities. The Solution of Linear Systems: Methods of Successive Approximation. Direct Methods of Inversion. Proper Values and Vectors: Normalization and Reduction of the Matrix. Proper Values and Vectors: Successive Approximation.

L. Fox (1964). *An Introduction to Numerical Linear Algebra*, Oxford University Press, Oxford, England.

Introduction, Matrix Algebra. Elimination Methods of Gauss, Jordan, and Aitken. Compact Elimination Methods of Doolittle, Crout, Banachiewicz, and Cholesky. Orthogonalization Methods. Condition, Accuracy, and Precision. Comparison of Methods, Measure of Work. Iterative and Gradient Methods. Iterative methods for Latent Roots and Vectors. Transformation Methods for Latent Roots and Vectors. Notes on Error Analysis for Latent Roots and Vectors.

J.H. Wilkinson (1965). *The Algebraic Eigenvalue Problem*, Clarendon Press, Oxford, England.

Theoretical Background. Perturbation Theory. Error Analysis. Solution of Linear Algebraic Equations. Hermitian Matrices. Reduction of a General Matrix to Condensed Form. Eigenvalues of Matrices of Condensed Forms. The LR and QR Algorithms. Iterative Methods.

G.E. Forsythe and C. Moler (1967). *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, Englewood Cliffs, NJ.

Reader's Background and Purpose of Book. Vector and Matrix Norms. Diagonal Form of a Matrix Under Orthogonal Equivalence. Proof of Diagonal Form Theorem. Types of Computational Problems in Linear Algebra. Types of Matrices encountered in Practical Problems. Sources of Computational Problems of Linear Algebra. Condition of a Linear System. Gaussian Elimination and LU Decomposition. Need for Interchanging Rows. Scaling Equations and Unknowns. The Crout and Doolittle Variants. Iterative Improvement. Computing the Determinant. Nearly Singular Matrices. Algol 60 Program. Fortran, Extended Algol, and PL/I Programs. Matrix Inversion. An Example: Hilbert Matrices. Floating Point Round-Off Analysis. Rounding Error in Gaussian Elimination. Convergence of Iterative Improvement. Positive Definite Matrices; Band Matrices. Iterative Methods for Solving Linear Systems. Nonlinear Systems of Equations.

Introductory (General)

A.R. Gourlay and G.A. Watson (1973). *Computational Methods for Matrix Eigenproblems*, John Wiley & Sons, New York.

Introduction. Background Theory. Reductions and Transformations. Methods for the Dominant Eigenvalue. Methods for the Subdominant Eigenvalue. Inverse Iteration. Jacobi's Methods. Givens and Householder's Methods. Eigensystem of a Symmetric Tridiagonal Matrix. The LR and QR Algorithms. Extensions of Jacobi's Method. Extension of Givens' and Householder's Methods. QR Algorithm for Hessenberg Matrices. generalized Eigenvalue Problems. Available Implementations.

G.W. Stewart (1973). *Introduction to Matrix Computations*, Academic Press, New York.

Preliminaries. Practicalities. The Direct Solution of Linear Systems. Norms, Limits, and Condition Numbers. The Linear Least Squares Problem. Eigenvalues and Eigenvectors. The QR Algorithm.

R.J. Goult, R.F. Hoskins, J.A. Milner and M.J. Pratt (1974). *Computational Methods in Linear Algebra*, John Wiley and Sons, New York.

Eigenvalues and Eigenvectors. Error Analysis. The Solution of Linear Equations by Elimination and Decomposition Methods. The Solution of Linear Systems of Equations by Iterative Methods. Errors in the Solution Sets of Equations. Computation of Eigenvalues and Eigenvectors. Errors in Eigenvalues and Eigenvectors. Appendix – A Survey of Essential Results from Linear Algebra.

T.F. Coleman and C.F. Van Loan (1988). *Handbook for Matrix Computations*, SIAM Publications, Philadelphia, PA.

Fortran 77, The Basic Linear Algebra Subprograms, Linpack, MATLAB.

W.W. Hager (1988). *Applied Numerical Linear Algebra*, Prentice-Hall, Englewood Cliffs, NJ.

Introduction. Elimination Schemes. Conditioning. Nonlinear Systems. Least Squares. Eigenproblems. Iterative Methods.

P.G. Ciarlet (1989). *Introduction to Numerical Linear Algebra and Optimisation*, Cambridge University Press.

A Summary of Results on Matrices. General Results in the Numerical Analysis of Matrices. Sources of Problems in the Numerical Analysis of Matrices. Direct Methods for the Solution of Linear Systems. Iterative Methods for the Solution of Linear Systems. Methods for the Calculation of Eigenvalues and Eigenvectors. A Review of Differential Calculus. Some Applications. General Results on Optimization. Some Algorithms. Introduction to Nonlinear Programming. Linear Programming.

D.S. Watkins (1991). *Fundamentals of Matrix Computations*, John Wiley and Sons, New York.

Gaussian Elimination and Its Variants. Sensitivity of Linear Systems; Effects of Roundoff Errors. Orthogonal Matrices and the Least-Squares Problem. Eigenvalues and Eigenvectors I. Eigenvalues and Eigenvectors II. Other Methods for the Symmetric Eigenvalue Problem. The Singular Value Decomposition.

P. Gill, W. Murray, and M.H. Wright (1991). *Numerical Linear Algebra and Optimization, Vol. 1*, Addison-Wesley, Reading, MA.

Introduction. Linear Algebra Background. Computation and Condition. Linear Equations. Compatible Systems. Linear Least Squares. Linear Constraints I: Linear Programming. The Simplex Method.

A. Jennings and J.J. McKeowen (1992). *Matrix Computation (2nd ed)*, John Wiley and Sons, New York.

Basic Algebraic and Numerical Concepts. Some Matrix Problems. Computer Implementation. Elimination Methods for Linear Equations. Sparse Matrix Elimination. Some Matrix Eigenvalue Problems. Transformation Methods for Eigenvalue Problems. Sturm Sequence Methods. Vector Iterative Methods for Partial Eigensolution. Orthogonalization and Re-Solution Techniques for Linear Equations. Iterative Methods for Linear Equations. Non-linear Equations. Parallel and Vector Computing.

B.N. Datta (1995). *Numerical Linear Algebra and Applications*. Brooks/Cole Publishing Company, Pacific Grove, California.

Review of Required Linear Algebra Concepts. Floating Point Numbers and Errors in Computations. Stability of Algorithms and Conditioning of Problems. Numerically Effective Algorithms and Mathematical Software. Some Useful Transformations in Numerical Linear Algebra and Their Applications. Numerical Matrix Eigenvalue Problems. The Generalized Eigenvalue Problem. The Singular Value Decomposition. A Taste of Roundoff Error Analysis.

M.T. Heath (1997). *Scientific Computing: An Introductory Survey*, McGraw-Hill, New York.

Scientific Computing. Systems of Linear Equations. Linear Least Squares. Eigenvalues and Singular Values. Nonlinear Equations. Optimization. Interpolation. Numerical Integration and Differentiation. Initial Value Problems for ODEs. Boundary Value Problems for ODEs. Partial Differential Equations. Fast Fourier Transform. Random Numbers and Simulation.

C.F. Van Loan (1997). *Introduction to Scientific Computing: A Matrix-Vector Approach Using Matlab*, Prentice Hall, Upper Saddle River, NJ.

Power Tools of the Trade. Polynomial Interpolation. Piecewise Polynomial Interpolation. Numerical Integration. Matrix Computations. Linear Systems. The QR and Cholesky Factorizations. Nonlinear Equations and Optimization. The Initial Value Problem.

Advanced (General)

N.J. Higham (1996). *Accuracy and Stability of Numerical Algorithms*, SIAM Publications, Philadelphia, PA.

Principles of Finite Precision Computation. Floating Point Arithmetic. Basics. Summation. Polynomials. Norms. Perturbation Theory for Linear Systems. Triangular Systems. LU Factorization and Linear Equations. Cholesky Factorization. Iterative Refinement. Block LU Factorization. Matrix Inversion. Condition Number Estimation. The Sylvester Equation. Stationary Iterative Methods. Matrix Powers. QR Factorization. The Least Squares Problem. Underdetermined Systems. Vandermonde Systems. Fast Matrix Multiplication. The Fast Fourier Transform and Applications. Automatic Error Analysis. Software Issues in Floating Point Arithmetic. A Gallery of Test Matrices.

J.W. Demmel (1996). *Numerical Linear Algebra*, SIAM Publications, Philadelphia, PA.

Introduction. Linear Equation Solving. Linear Least Squares Problems. Nonsymmetric Eigenvalue Problems. The Symmetric Eigenproblem and Singular Value Decomposition. Iterative Methods for Linear Systems and Eigenvalue Problems. Iterative Algorithms for Eigenvalue Problems.

L.N. Trefethen and D. Bau III (1997). *Numerical Linear Algebra*, SIAM Publications, Philadelphia, PA.

Matrix-Vector Multiplication. Orthogonal Vectors and Matrices. Norms. The Singular Value Decomposition. More on the SVD. Projectors. QR Factorization. Gram-Schmidt Orthogonalization. MATLAB. Householder Triangularization. Least-Squares Problems. Conditioning and Condition Numbers. Floating Point Arithmetic. Stability. More on Stability. Stability of Householder Triangularization. Stability of Back Substitution. Conditioning of Least-Squares Problems. Stability of Least-Squares Algorithms. Gaussian Elimination. Pivoting. Stability of Gaussian Elimination. Cholesky Factorization. Eigenvalue Problems. Overview of Eigenvalue Algorithms. Reduction to Hessenberg/Tridiagonal Form. Rayleigh Quotient, Inverse Iteration. QR Algorithm Without Shifts. QR Algorithm With Shifts. Other Eigenvalue Algorithms. Computing the SVD. Overview of Iterative Methods. The Arnoldi Iteration. How Arnoldi Locates Eigenvalues. GMRES. The Lanczos Iteration. Orthogonal Polynomials and Gauss Quadrature. Conjugate Gradients. Biorthogonalization Methods. Preconditioning. The Definition of Numerical Analysis.

Analytical

F.R. Gantmacher (1959). *The Theory of Matrices Vol. 1*, Chelsea, New York.

Matrices and Operations on Matrices. The Algorithm of Gauss and Some of its Applications. Linear Operators in an n-dimensional Vector Space. The Characteristic Polynomial and the Minimum Polynomial of a Matrix. Functions of Matrices, Equivalent Transformations of Polynomial Matrices, Analytic Theory of Elementary Divisors. The Structure of a Linear Operator in an n-dimensional Space. Matrix Equations. Linear Operators in a Unitary Space. Quadratic and Hermitian Forms.

F.R. Gantmacher (1959). *The Theory of Matrices Vol. 2*, Chelsea, New York.

Complex Symmetric, Skew-Symmetric, and Orthogonal Matrices. Singular Pencils of Matrices. Matrices with Nonnegative Elements. Application of the Theory of Matrices to the Investigation of Systems of Linear Differential Equations. The Problem of Routh-Hurwitz and Related Questions.

A. Berman and R.J. Plemmons (1979). *Nonnegative Matrices in the Mathematical Sciences*, Academic Press, New York. Reprinted with additions in 1994 by SIAM Publications, Philadelphia, PA.

Matrices Which Leave a Cone Invariant. Nonnegative Matrices. Semigroups of Nonnegative Matrices. Symmetric Nonnegative Matrices. Generalized Inverse-Positivity. M-Matrices. Iterative Methods for Linear Systems. Finite Markov Chains. Input-Output Analysis in Economics. The Linear Complementarity Problem.

G.W. Stewart and J. Sun (1990). *Matrix Perturbation Theory*, Academic Press, San Diego.

Preliminaries. Norms and Metrics. Linear Systems and Least Squares Problems. The Perturbation of Eigenvalues. Invariant Subspaces. Generalized Eigenvalue Problems.

R. Horn and C. Johnson (1985). *Matrix Analysis*, Cambridge University Press, New York.

Review and Miscellanea. Eigenvalues, Eigenvectors, and Similarity. Unitary Equivalence and Normal Matrices. Canonical Forms. Hermitian and Symmetric Matrices. Norms for Vectors and Matrices. Location and Perturbation of Eigenvalues. Positive Definite Matrices.

R. Horn and C. Johnson (1991). *Topics in Matrix Analysis*, Cambridge University Press, New York.

The Field of Values. Stable Matrices and Inertia. Singular Value Inequalities. Matrix Equations and the Kronecker Product. The Hadamard Product. Matrices and Functions.

Linear Equation Problems

D.M. Young (1971). *Iterative Solution of Large Linear Systems*, Academic Press, New York.

Introduction. Matrix Preliminaries. Linear Stationary Iterative Methods. Convergence of the Basic Iterative Methods. Eigenvalues of the SOR Method for Consistently Ordered Matrices. Determination of the Optimum Relaxation Parameter. Norms of the SOR Method. The Modified SOR Method: Fixed Parameters. Nonstationary Linear Iterative Methods. The Modified SOR Method: Variable Parameters. Semi-Iterative Methods. Extensions of the SOR Theory; Stieltjes Matrices. Generalized Consistently Ordered Matrices. Group Iterative Methods. Symmetric SOR Method and Related Methods. Second Degree Methods. Alternating Direction Implicit Methods. Selection of an Iterative Method.

L.A. Hageman and D.M. Young (1981). *Applied Iterative Methods*, Academic Press, New York.

Background on Linear Algebra and Related Topics. Background on Basic Iterative Methods. Polynomial Acceleration. Chebyshev Acceleration. An Adaptive Chebyshev Procedure Using Special Norms. Adaptive Chebyshev Acceleration. Conjugate Gradient Acceleration. Special Methods for Red/Black Partitionings. Adaptive Procedures for Successive Overrelaxation Method. The Use of Iterative Methods in the Solution of Partial Differential Equations. Case Studies. The Nonsymmetrizable Case.

A. George and J. W-H. Liu (1981). *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall Inc., Englewood Cliffs, New Jersey.

Introduction. Fundamentals. Some Graph Theory Notation and Its Use in the Study of Sparse Symmetric Matrices. BAnd and Envelope Methods. General Sparse Methods. Quotient Tree Methods for Finite Element and Finite Difference Problems. One-Way Dissection Methods for Finite Element Problems. Nested Dissection Methods. Numerical Experiments.

S. Pissanetsky (1984). *Sparse Matrix Technology*, Academic Press, New York.

Fundamentals. Linear Algebraic Equations. Numerical Errors in Gaussian Elimination. Ordering for Gauss Elimination: Symmetric Matrices. Ordering for Gauss Elimination: General Matrices. Sparse Eigenanalysis. Sparse Matrix Algebra. Connectivity and Nodal Assembly. General Purpose Algorithms.

I.S. Duff, A.M. Erisman, and J.K. Reid (1986). *Direct Methods for Sparse Matrices*, Oxford University Press, New York.

Introduction. Sparse Matrices:Storage Schemes and Simple Operations. Gaussian Elimination for Dense Matrices: The Algebraic Problem. Gaussian Elimination for Dense Matrices: Numerical Considerations. Gaussian Elimination for Sparse Matrices: An Introduction. Reduction to Block Triangular Form. Local Pivotal Strategies for Sparse Matrices. Ordering Sparse Matrices to Special Forms. Implementing Gaussian Elimination: Analyse with Numerical Values. Implementing Gaussian Elimination with Symbolic Analyse. Partitioning, Matrix Modification, and Tearing. Other Sparsity-Oriented Issues.

R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. van der Vorst (1993). *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM Publications, Philadelphia, PA.

Introduction. Why Use Templates? What Methods are Covered? Iterative Methods. Stationary Methods. Nonstationary Iterative Methods. Survey of Recent Krylov Methods. Jacobi, Incomplete, SSOR, and Polynomial Preconditioners. Complex Systems. Stopping Criteria. Data Structures. Parallelism. The Lanczos Connection. Block Iterative Methods. Reduced System Preconditioning. Domain Decomposition Methods. Multigrid Methods. Row Projection Methods.

W. Hackbusch (1994). *Iterative Solution of Large Sparse Systems of Equations*, Springer-Verlag, New York.

Introduction. Recapitulation of Linear Algebra. Iterative Methods. Methods of Jacobi and Gauss-Seidel and SOR Iteration in the Positive Definite Case. Analysis in the 2-Cyclic Case. Analysis for M-Matrices. Semi-Iterative Methods. Transformations, Secondary Iterations, Incomplete Triangular Decompositions. Conjugate Gradient Methods. Multi-Grid Methods. Domain Decomposition Methods.

O. Axelsson (1994). *Iterative Solution Methods*, Cambridge University Press.

Direct Solution Methods. Theory of Matrix Eigenvalues. Positive Definite Matrices, Schur Complements, and Generalized Eigenvalue Probems. Reducible and Irreducible Matrices and the Perron-Frobenious Theory for Nonnegative Matrices. Basic Iterative Methods and Their Rates of Convergence. M-Matrices, Convergent Splittings, and the SOR Method. Incomplete Factorization Preconditioning Methods. Approximate Matrix Inverses and Corresponding Preconditioning Methods. Block Diagonal and Schur Complement Preconditionngs. Estimates of Eigenvalues and Condition Numbers for Preconditioned Matrices. Conjugate Gradient and Lanczos-Type Methods. Generalized Conjugate Gradient Methods. The Rate of Convergence of the Conjugate Gradient Method.

Y. Saad (1996). *Iterative Methods for Sparse Linear Systems*, PWS Publishing Co., Boston.

Background in Linear Algebra. Discretization of PDEs. Sparse Matrices. Basic Iterative Methods. Projection Methods. Krylov Subspace Methods – Part I. Krylov Subspace Methods – Part II. Methods Related to the Normal Equations. Preconditioned Iterations. Preconditioning Techniques. Parallel Implementations. Parallel Preconditioners. Domain Decomposition Methods.

Linear Fitting Problems

C.L. Lawson and R.J. Hanson (1974). *Solving Least Squares Problems*, Prentice-Hall, Englewood Cliffs, NJ. Reprinted with a detailed "new developments" appendix in 1996 by SIAM Publications, Philadelphia, PA.

Introduction. Analysis of the Least Squares Problem. Orthogonal Decomposition by Certain Elementary Transformations. Orthogonal Decomposition by Singular Value Decomposition. Perturbation Theorems for Singular Values. Bounds for the Condition Number of a Triangular Matrix. The Pseudoinverse. Perturbation Bounds for the Pseudoinverse. Perturbation Bounds for the Solution of Problem LS. Numerical Computations Using Elementary Orthogonal Transformations. Computing the Solution for the Overdetermined or Exactly Determined Full Rank Problem. Computation of the Covariance Matrix of the Solution Parameters. Computing the Solution for the Underdetermined Full Rank Problem. Computing the Solution for Problem LS with Possibly Deficient Pseudorank. Analysis of Computing Errors for Householder Transformations. Analysis of Computing Errors for the Problem LS. Analysis of Computing Errors for the Problem LS Using Mixed Precision Arithmetic. Computation of the Singular Value Decomposition and the Solution of Problem LS. Other Methods for Least Squares Problems. Linear Least Squares with Linear Equality Constraints Using a Basis of the Null Space. Linear Least Squares with Linear Equality Constraints by Direct Elimination. Linear Least Squares with Linear Equality Constraints by Weighting. Linear least Squares with Linear Inequality Constraints. Modifying a QR Decomposition to Add or Remove Column Vectors. Practical Analysis of Least Squares Problems. Examples of Some Methods of Analyzing a Least Squares Problem. Modifying a QR Decomposition to Add or Remove Row Vectors with Application to Sequential Processing of Problems Having a Large or Banded Coefficient Matrix.

R.W. Farebrother (1987). *Linear Least Squares Computations*, Marcel Dekker, New York.

The Gauss and Gauss-Jordan Methods. Matrix Analysis of Gauss's Method: The Cholesky and Doolittle Decompositions. The Linear Algebraic Model: The Method of Averages and the Method of Least Squares. The Cauchy-Bienayme, Laplace, and Schmidt Procedures. Householder Procedures. Givens Procedures. Updating the QU Decomposition. Pseudorandom Numbers. The Standard Linear Model. Condition Numbers. Instrumental Variable Estimators. Generalized Least Squares Estimation. Iterative Solutions of Linear and Nonlinear Least Squares Problems. Canonical Expressions for the Least Squares Estimators and Test Statistics. Traditional Expressions for the Least Squares Updating Formulas and test Statistics. Least Squares Estimation Subject to Linear Constraints.

S. Van Huffel and J. Vandewalle (1991). *The Total Least Squares Problem: Computational Aspects and Analysis*, SIAM Publications, Philadelphia, PA.

Introduction. Basic Principles of the Total Least Squares Problem. Extensions of the Basic Total Least Squares Problem. Direct Speed Improvement of the Total Least Squares Computations. Iterative Speed Improvement for Solving Slowly Varying Total Least Squares Problems. Algebraic Connections Between Total Least Squares and Least Squares Problems. Sensitivity Analysis of Total Least Squares and Least Squares Problems in the Presence of Errors in All Data. Statistical Properties of the Total Least Squares Problem. Algebraic Connections Between Total Least Squares Estimation and Classical Linear Regression in Multicollinearity Problems. Conclusions.

Å. Björck (1996). *Numerical Methods for Least Squares Problems*, SIAM Publications, Philadelphia, PA.

Mathematical and Statistical Properties of Least Squares Solutions. Basic Numerical Methods. Modified Least Squares Problems. Generalized Least Squares Problems. Constrained Least Squares Problems. Direct Methods for Sparse Least Squares Problems. Iterative Methods for Least Squares Problems. Least Squares with Special Bases. Nonlinear Least Squares Problems.

Eigenvalue Problems

B.N. Parlett (1980). *The Symmetric Eigenvalue Problem*, Prentice-Hall, Englewood Cliffs, NJ.

Basic Facts about Self-Adjoint Matrices. Tasks, Obstacles, and Aids. Counting Eigenvalues. Simple Vector Iterations. Deflation. Useful Orthogonal Matrices. Tridiagonal Form. The QL and QR Algorithms. Jacobi Methods. Eigenvalue Bounds. Approximation from a Subspace. Krylov Subspaces. Lanczos Algorithms. Subspace Iteration. The General Linear Eigenvalue Problem.

J. Cullum and R.A. Willoughby (1985a). *Lanczos Algorithms for Large Symmetric Eigenvalue Computations, Vol. I Theory*, Birkhaüser, Boston.

Preliminaries: Notation and Definitions. Real Symmetric Problems. Lanczos Procedures, Real Symmetric Problems. Tridiagonal Matrices. Lanczos Procedures with No Reorthogonalization for Symmetric Problems. Real Rectangular Matrices. Non-Defective Complex Symmetric Matrices. Block Lanczos Procedures, Real Symmetric Matrices.

J. Cullum and R.A. Willoughby (1985b). *Lanczos Algorithms for Large Symmetric Eigenvalue Computations, Vol. II Programs*, Birkhaüser, Boston.

Lanczos Procedures. Real Symmetric Matrices. Hermitian Matrices. Factored Inverses of Real Symmetric Matrices. Real Symmetric Generalized Problems. Real Rectangular Problems. Nondefective Complex Symmetric Matrices. Real Symmetric Matrices, Block Lanczos Code. Factored Inverses, Real Symmetric Matrices, Block Lanczos Code.

Y. Saad (1992). *Numerical Methods for Large Eigenvalue Problems: Theory and Algorithms*, John Wiley and Sons, New York.

Background in Matrix Theory and Linear Algebra. Perturbation Theory and Error Analysis. The Tools of Spectral Approximation. Subspace Iteration. Krylov Subspace Methods. Acceleration Techniques and Hybrid Methods. Preconditioning Techniques. Non-Standard Eigenvalue Problems. Origins of Matrix Eigenvalue Problems.

F. Chatelin (1993). *Eigenvalues of Matrices*, John Wiley and Sons, New York.

Supplements from Linear Algebra. Elements of Spectral Theory. Why Compute Eigenvalues. Error Analysis. Foundations of Methods for Computing Eigenvalues. Numerical Methods for Large Matrices. Chebyshev's Iterative Methods.

High Performance

W. Schönauer (1987). *Scientific Computing on Vector Computers*, North Holland, Amsterdam.

Introduction. The First Commercially Significant Vector Computer. The Arithmetic Performance of the First Commercially Significant Vector Computer. Hockney's $n_{1/2}$ and Timing Formulae. Fortran and Autovectorization. Behavior of Programs. Some Basic Algorithms, Recurrences. Matrix Operations. Systems of Linear Equations with Full Matrices. Tridiagonal Linear Systems. The Iterative Solution of Linear Equations. Special Applications. The Fujitsu VPs and Other Japanese Vector Computers. The Cray-2. The IBM VF and Other Vector Processors. The Convex C1.

R.W. Hockney and C.R. Jesshope (1988). *Parallel Computers 2*, Adam Hilger, Bristol and Philadelphia.

Introduction. Pipelined Computers. Processor Arrays. Parallel Languages. Parallel Algorithms. Future Developments.

J.J. Modi (1988). *Parallel Algorithms and Matrix Computation*, Oxford University Press, Oxford.

General Principles of Parallel Computing. Parallel Techniques and Algorithms. Parallel Sorting Algorithms. Solution of a System of Linear Algebraic Equations. The Symmetric Eigenvalue Problem: Jacobi's Method. QR Factorization. Singular Value Decomposition and Related Problems.

J. Ortega (1988). *Introduction to Parallel and Vector Solution of Linear Systems*, Plenum Press, New York.

Introduction. Direct Methods for Linear Equations. Iterative Methods for Linear Equations.

J. Dongarra, I. Duff, D. Sorensen, and H. van der Vorst (1990). *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM Publications, Philadelphia, PA.

Vector and Parallel Processing. Overview of Current High-Performance Computers. Implementation Details and Overhead. Performance Analysis, Modeling, and Measurements. Building Blocksin Linear Algebra. Direct Solution of Sparse Linear Systems. Iterative Solution of Sparse Linear Systems.

Y. Robert (1990). *The Impact of Vector and Parallel Architectures on the Gaussian Elimination Algorithm*, Halsted Press, New York.

Introduction. Vector and Parallel Architectures. Vector Multiprocessor Computing. Hypercube Computing. Systolic Computing. Task Graph Scheduling. Analysis of Distributed Algorithms. Design Methodologies.

G.H. Golub and J.M. Ortega (1993). *Scientific Computing: An Introduction with Parallel Computing*, Academic Press, Boston.

The World of Scientific Computing. Linear Algebra. Parallel and Vector Computing. Polynomial Approximation. Continuous Problems Solved Discretely. Direct Solution of Linear Equations. Parallel Direct Methods. Iterative Methods. Conjugate Gradient-Type Methods.

Edited Volumes

D.J. Rose and R. A. Willoughby, eds. (1972). *Sparse Matrices and Their Applications*, Plenum Press, New York, 1972

J.R. Bunch and D.J. Rose, eds. (1976). *Sparse Matrix Computations*, Academic Press, New York.

I.S. Duff and G.W. Stewart, eds. (1979). *Sparse Matrix Proceedings, 1978*, SIAM Publications, Philadelphia, PA.

I.S. Duff, ed. (1981). *Sparse Matrices and Their Uses*, Academic Press, New York.

Å. Björck, R.J. Plemmons, and H. Schneider, eds. (1981). *Large-Scale Matrix Problems*, North-Holland, New York.

G. Rodrigue, ed. (1982). *Parallel Computation*, Academic Press, New York.

B. Kågström and A. Ruhe, eds. (1983). *Matrix Pencils*, Proc. Pite Havsbad, 1982, Lecture Notes in Mathematics 973, Springer-Verlag, New York and Berlin.

J. Cullum and R.A. Willoughby, eds. (1986). *Large Scale Eigenvalue Problems*, North-Holland, Amsterdam.

A. Wouk, ed. (1986). *New Computing Environments: Parallel, Vector, and Systolic, SIAM Publications*, Philadelphia, PA.

M.T. Heath, ed. (1986). *Proceedings of First SIAM Conference on Hypercube Multiprocessors*, SIAM Publications, Philadelphia, PA.

M.T. Heath, ed. (1987). *Hypercube Multiprocessors*, SIAM Publications, Philadelphia, PA.

G. Fox, ed. (1988). *The Third Conference on Hypercube Concurrent Computers and Applications, Vol. II – Applications*, ACM Press, New York.

M.H. Schultz, ed. (1988). *Numerical Algorithms for Modern Parallel Computer Architectures*, IMA Volumes in Mathematics and Its Applications, Number 13, Springer-Verlag, Berlin.

E.F. Deprettere, ed. (1988). *SVD and Signal Processing*. Elsevier, Amsterdam.

B.N. Datta, C.R. Johnson. M.A. Kaashoek, R. Plemmons, and E.D. Sontag, eds. (1988), *Linear Algebra in Signals, Systems, and Control*, SIAM Publications, Philadelphia, PA.

J. Dongarra, I. Duff, P. Gaffney, and S. McKee, eds. (1989), *Vector and Parallel Computing*, Ellis Horwood, Chichester, England.

O. Axelsson, ed. (1989). "Preconditioned Conjugate Gradient Methods," *BIT 29:4*.

K. Gallivan, M. Heath, E. Ng, J. Ortega, B. Peyton, R. Plemmons, C. Romine, A. Sameh, and B. Voigt (1990), *Parallel Algorithms for Matrix Computations*, SIAM Publications, Philadelphia, PA.

G.H. Golub and P. Van Dooren, eds. (1991). *Numerical Linear Algebra, Digital Signal Processing, and Parallel Algorithms*. Springer-Verlag, Berlin.

R. Vaccaro, ed. (1991). *SVD and Signal Processing II: Algorithms, Analysis, and Applications*. Elsevier, Amsterdam.

R. Beauwens and P. de Groen, eds. (1992). *Iterative Methods in Linear Algebra*, Elsevier (North-Holland), Amsterdam.

R.J. Plemmons and C.D. Meyer, eds. (1993). *Linear Algebra, Markov Chains, and Queuing Models*, Springer-Verlag, New York.

M.S. Moonen, G.H. Golub, and B.L.R. de Moor, eds. (1993). *Linear Algebra for Large Scale and Real-Time Applications*, Kluwer, Dordrecht, The Netherlands.

J.D. Brown, M.T. Chu, D.C. Ellison, and R.J. Plemmons, eds. (1994). *Proceedings of the Cornelius Lanczos International Centenary Conference*, SIAM Publications, Philadelphia, PA.

R.V. Patel, A.J. Laub, and P.M. Van Dooren, eds. (1994). *Numerical Linear Algebra Techniques for Systems and Control*, IEEE Press, Piscataway, New Jersey.

J. Lewis, ed. (1994). *Proceedings of the Fifth SIAM Conference on Applied Linear Algebra*, SIAM Publications, Philadelphia, PA.

A. Bojanczyk and G. Cybenko, eds. (1995). *Linear Algebra for Signal Processing*, IMA Volumes in Mathematics and Its Applications, Springer-Verlag, New York.

M. Moonen and B. De Moor, eds. (1995). *SVD and Signal Processing III: Algorithms, Analysis, and Applications*. Elsevier, Amsterdam.

Matrix Computations

Chapter 1

Matrix Multiplication Problems

§1.1 Basic Algorithms and Notation

§1.2 Exploiting Structure

§1.3 Block Matrices and Algorithms

§1.4 Vectorization and Re-Use Issues

The proper study of matrix computations begins with the study of the matrix-matrix multiplication problem. Although this problem is simple mathematically it is very rich from the computational point of view. We begin in §1.1 by looking at the several ways that the matrix multiplication problem can be organized. The "language" of partitioned matrices is established and used to characterize several linear algebraic "levels" of computation.

If a matrix has structure, then it is usually possible to exploit it. For example, a symmetric matrix can be stored in half the space as a general matrix. A matrix-vector product that involves a matrix with many zero entries may require much less time to execute than a full matrix times a vector. These matters are discussed in §1.2.

In §1.3 block matrix notation is established. A block matrix is a matrix with matrix entries. This concept is very important from the standpoint of both theory and practice. On the theoretical side, block matrix notation allows us to prove important matrix factorizations very succinctly. These factorizations are the cornerstone of numerical linear algebra. From the computational point of view, block algorithms are important because they

are rich in matrix multiplication, the operation of choice for many new high performance computer architectures.

These new architectures require the algorithm designer to pay as much attention to memory traffic as to the actual amount of arithmetic. This aspect of scientific computation is illustrated in §1.4 where the critical issues of vector pipeline computing are discussed: stride, vector length, the number of vector loads and stores, and the level of vector re-use.

Before You Begin

It is important to be familiar with the MATLAB language. See the texts by Pratap(1995) and Van Loan (1996). A richer introduction to high performance matrix computations is given in Dongarra, Duff, Sorensen, and Duff (1991). This chapter's LAPACK connections include

LAPACK: Some General Operations		
_SCAL	$x \leftarrow \alpha x$	Vector scale
DOT	$\mu \leftarrow x^T y$	Dot product
_AXPY	$y \leftarrow \alpha x + y$	Saxpy
_GEMV	$y \leftarrow \alpha Ax + \beta y$	Matrix-vector multiplication
_GER	$A \leftarrow A + \alpha xy^T$	Rank-1 update
_GEMM	$C \leftarrow \alpha AB + \beta C$	Matrix multiplication

LAPACK: Some Symmetric Operations		
_SYMV	$y \leftarrow \alpha Ax + \beta y$	Matrix-vector multiplication
_SPMV	$y \leftarrow \alpha Ax + \beta y$	Matrix-vector multiplication (Packed)
_SYR	$A \leftarrow \alpha xx^T + A$	Rank-1 update
_SYR2	$A \leftarrow \alpha xy^T + \alpha yx^T + A$	Rank-2 update
_SYRK	$C \leftarrow \alpha AA^T + \beta C$	Rank-k update
_SYR2K	$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$	Rank-$2k$ update
_SYMM	$C = \alpha AB + \beta C$ or $(\alpha BA + \beta C)$	Symmetric/General Product

LAPACK: Some Band/Triangular Operations		
_GBMV	$y \leftarrow \alpha Ax + \beta y$	General Band
_SBMV	$y \leftarrow \alpha Ax + \beta y$	Symmetric Band
_TBMV	$x \leftarrow \alpha Ax$	Triangular
_TPMV	$x \leftarrow \alpha Ax$	Triangular Packed
_TRMM	$B \leftarrow \alpha AB$ (or BA)	Triangular/General Product

1.1 Basic Algorithms and Notation

Matrix computations are built upon a hierarchy of linear algebraic operations. Dot products involve the scalar operations of addition and multiplication. Matrix-vector multiplication is made up of dot products. Matrix-matrix multiplication amounts to a collection of matrix-vector products. All of these operations can be described in algorithmic form or in the language of linear algebra. Our primary objective in this section is to show

how these two styles of expression complement each another. Along the way we pick up notation and acquaint the reader with the kind of thinking that underpins the matrix computation area. The discussion revolves around the matrix multiplication problem, a computation that can be organized in several ways.

1.1.1 Matrix Notation

Let $\mathbb{R}$ denote the set of real numbers. We denote the vector space of all m-by-n real matrices by $\mathbb{R}^{m\times n}$:

$$A \in \mathbb{R}^{m\times n} \iff A = (a_{ij}) = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \quad a_{ij} \in \mathbb{R}.$$

If a capital letter is used to denote a matrix (e.g. A, B, Δ), then the corresponding lower case letter with subscript ij refers to the (i,j) entry (e.g., a_{ij} , b_{ij}, δ_{ij}). As appropriate, we also use the notation $[A]_{ij}$ and $A(i,j)$ to designate the matrix elements.

1.1.2 Matrix Operations

Basic matrix operations include *transposition* ($\mathbb{R}^{m\times n} \to \mathbb{R}^{n\times m}$),

$$C = A^T \qquad \Longrightarrow \qquad c_{ij} = a_{ji},$$

addition ($\mathbb{R}^{m\times n} \times \mathbb{R}^{m\times n} \to \mathbb{R}^{m\times n}$),

$$C = A + B \qquad \Longrightarrow \qquad c_{ij} = a_{ij} + b_{ij},$$

scalar-matrix multiplication, ($\mathbb{R} \times \mathbb{R}^{m\times n} \to \mathbb{R}^{m\times n}$),

$$C = \alpha A \qquad \Longrightarrow \qquad c_{ij} = \alpha a_{ij},$$

and *matrix-matrix multiplication* ($\mathbb{R}^{m\times p} \times \mathbb{R}^{p\times n} \to \mathbb{R}^{m\times n}$),

$$C = AB \qquad \Longrightarrow \qquad c_{ij} = \sum_{k=1}^{r} a_{ik}b_{kj}.$$

These are the building blocks of matrix computations.

1.1.3 Vector Notation

Let $\mathbb{R}^n$ denote the vector space of real n-vectors:

$$x \in \mathbb{R}^n \iff x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \quad x_i \in \mathbb{R}.$$

We refer to x_i as the ith component of x. Depending upon context, the alternative notations $[x]_i$ and $x(i)$ are sometimes used.

Notice that we are identifying $\mathbb{R}^n$ with $\mathbb{R}^{n\times 1}$ and so the members of $\mathbb{R}^n$ are *column* vectors. On the other hand, the elements of $\mathbb{R}^{1\times n}$ are *row* vectors:

$$x \in \mathbb{R}^{1\times n} \iff x = (x_1, \ldots, x_n).$$

If x is a column vector, then $y = x^T$ is a row vector.

1.1.4 Vector Operations

Assume $a \in \mathbb{R}$, $x \in \mathbb{R}^n$, and $y \in \mathbb{R}^n$. Basic vector operations include *scalar-vector multiplication*,

$$z = ax \qquad \Longrightarrow \qquad z_i = ax_i,$$

vector addition,

$$z = x + y \qquad \Longrightarrow \qquad z_i = x_i + y_i,$$

the *dot product* (or *inner product*),

$$c = x^T y \qquad \Longrightarrow \qquad c = \sum_{i=1}^{n} x_i y_i,$$

and *vector multiply* (or the *Hadamard product*)

$$z = x.*y \qquad \Longrightarrow \qquad z_i = x_i y_i.$$

Another very important operation which we write in "update form" is the *saxpy*:

$$y = ax + y \quad \Longrightarrow \quad y_i = ax_i + y_i$$

Here, the symbol "=" is being used to denote assignment, not mathematical equality. The vector y is being updated. The name "saxpy" is used in LAPACK, a software package that implements many of the algorithms in this book. One can think of "saxpy" as a mnemonic for "scalar a x plus y."

1.1.5 The Computation of Dot Products and Saxpys

We have chosen to express algorithms in a stylized version of the MATLAB language. MATLAB is a powerful interactive system that is ideal for matrix computation work. We gradually introduce our stylized MATLAB notation in this chapter beginning with an algorithm for computing dot products.

Algorithm 1.1.1 (Dot Product) If $x, y \in \mathbb{R}^n$, then this algorithm computes their dot product $c = x^T y$.

```
c = 0
for i = 1:n
    c = c + x(i)y(i)
end
```

The dot product of two n-vectors involves n multiplications and n additions. It is an "$O(n)$" operation, meaning that the amount of work is linear in the dimension. The saxpy computation is also an $O(n)$ operation, but it returns a vector instead of a scalar.

Algorithm 1.1.2 (Saxpy) If $x, y \in \mathbb{R}^n$ and $a \in \mathbb{R}$, then this algorithm overwrites y with $ax + y$.

```
for i = 1:n
    y(i) = ax(i) + y(i)
end
```

It must be stressed that the algorithms in this book are encapsulations of critical computational ideas and not "production codes."

1.1.6 Matrix-Vector Multiplication and the Gaxpy

Suppose $A \in \mathbb{R}^{m \times n}$ and that we wish to compute the update

$$y = Ax + y$$

where $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$ are given. This generalized saxpy operation is referred to as a *gaxpy*. A standard way that this computation proceeds is to update the components one at a time:

$$y_i = \sum_{j=1}^{n} a_{ij} x_j + y_i \qquad i = 1{:}m.$$

This gives the following algorithm.

Algorithm 1.1.3 (Gaxpy: Row Version) If $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$, and $y \in \mathbb{R}^m$, then this algorithm overwrites y with $Ax + y$.


```
for i = 1:m
    for j = 1:n
        y(i) = A(i,j)x(j) + y(i)
    end
end
```

An alternative algorithm results if we regard Ax as a linear combination of A's columns, e.g.,

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \begin{bmatrix} 7 \\ 8 \end{bmatrix} = \begin{bmatrix} 1\cdot 7 + 2\cdot 8 \\ 3\cdot 7 + 4\cdot 8 \\ 5\cdot 7 + 6\cdot 8 \end{bmatrix} = 7\begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} + 8\begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix} = \begin{bmatrix} 23 \\ 53 \\ 83 \end{bmatrix}.$$

Algorithm 1.1.4 (Gaxpy: Column Version) If $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$, and $y \in \mathbb{R}^m$, then this algorithm overwrites y with $Ax + y$.

```
for j = 1:n
    for i = 1:m
        y(i) = A(i,j)x(j) + y(i)
    end
end
```

Note that the inner loop in either gaxpy algorithm carries out a saxpy operation. The column version was derived by rethinking what matrix-vector multiplication "means" at the vector level, but it could also have been obtained simply by interchanging the order of the loops in the row version. In matrix computations, it is important to relate loop interchanges to the underlying linear algebra.

1.1.7 Partitioning a Matrix into Rows and Columns

Algorithms 1.1.3 and 1.1.4 access the data in A by row and by column respectively. To highlight these orientations more clearly we introduce the language of *partitioned matrices*.

From the row point of view, a matrix is a stack of row vectors:

$$A \in \mathbb{R}^{m \times n} \quad \Longleftrightarrow \quad A = \begin{bmatrix} r_1^T \\ \vdots \\ r_m^T \end{bmatrix} \quad r_k \in \mathbb{R}^n. \qquad (1.1.1)$$

This is called a *row partition* of A. Thus, if we row partition

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix},$$

then we are choosing to think of A as a collection of rows with

$$r_1^T = [\,1 \quad 2\,], \qquad r_2^T = [\,3 \quad 4\,], \quad \text{and} \quad r_3^T = [\,5 \quad 6\,].$$

With the row partitioning (1.1.1) Algorithm 1.1.3 can be expressed as follows:

for $i = 1{:}m$
 $y_i = r_i^T x + y(i)$
end

Alternatively, a matrix is a collection of column vectors:

$$A \in \mathbb{R}^{m\times n} \iff A = [\,c_1, \ldots, c_n\,], \quad c_k \in \mathbb{R}^m. \qquad (1.1.2)$$

We refer to this as a *column partition* of A. In the 3-by-2 example above, we thus would set c_1 and c_2 to be the first and second columns of A respectively:

$$c_1 = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} \qquad c_2 = \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix}.$$

With (1.1.2) we see that Algorithm 1.1.4 is a saxpy procedure that accesses A by columns:

for $j = 1{:}n$
 $y = x_j c_j + y$
end

In this context appreciate y as a running vector sum that undergoes repeated saxpy updates.

1.1.8 The Colon Notation

A handy way to specify a column or row of a matrix is with the "colon" notation. If $A \in \mathbb{R}^{m\times n}$, then $A(k,:)$ designates the kth row, i.e.,

$$A(k,:) = [a_{k1}, \ldots, a_{kn}].$$

The kth column is specified by

$$A(:,k) = \begin{bmatrix} a_{1k} \\ \vdots \\ a_{mk} \end{bmatrix}.$$

With these conventions we can rewrite Algorithms 1.1.3 and 1.1.4 as

for $i = 1{:}m$
 $y(i) = A(i,:)x + y(i)$
end

and

for $j = 1{:}n$
 $y = x(j)A(:,j) + y$
end

respectively. With the colon notation we are able to suppress iteration details. This frees us to think at the vector level and focus on larger computational issues.

1.1.9 The Outer Product Update

As a preliminary application of the colon notation, we use it to understand the *outer product update*

$$A = A + xy^T, \qquad A \in \mathbb{R}^{m\times n},\ x \in \mathbb{R}^m,\ y \in \mathbb{R}^n.$$

The outer product operation xy^T "looks funny" but is perfectly legal, e.g.,

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} [\,4 \quad 5\,] = \begin{bmatrix} 4 & 5 \\ 8 & 10 \\ 12 & 15 \end{bmatrix}.$$

This is because xy^T is the product of two "skinny" matrices and the number of columns in the left matrix x equals the number of rows in the right matrix y^T. The entries in the outer product update are prescribed by

for $i = 1{:}m$
 for $j = 1{:}n$
  $a_{ij} = a_{ij} + x_i y_j$
 end
end

The mission of the j loop is to add a multiple of y^T to the i-th row of A, i.e.,

for $i = 1{:}m$
 $A(i,:) = A(i,:) + x(i)y^T$
end

On the other hand, if we make the i-loop the inner loop, then its task is to add a multiple of x to the jth column of A:

for $j = 1{:}n$
 $A(:,j) = A(:,j) + y(j)x$
end

Note that both outer product algorithms amount to a set of saxpy updates.

1.1.10 Matrix-Matrix Multiplication

Consider the 2-by-2 matrix-matrix multiplication AB. In the dot product formulation each entry is computed as a dot product:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1\cdot 5 + 2\cdot 7 & 1\cdot 6 + 2\cdot 8 \\ 3\cdot 5 + 4\cdot 7 & 3\cdot 6 + 4\cdot 8 \end{bmatrix}.$$

In the saxpy version each column in the product is regarded as a linear combination of columns of A:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \left[5\begin{bmatrix} 1 \\ 3 \end{bmatrix} + 7\begin{bmatrix} 2 \\ 4 \end{bmatrix},\ 6\begin{bmatrix} 1 \\ 3 \end{bmatrix} + 8\begin{bmatrix} 2 \\ 4 \end{bmatrix} \right].$$

Finally, in the outer product version, the result is regarded as the sum of outer products:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix} [\,5 \quad 6\,] + \begin{bmatrix} 2 \\ 4 \end{bmatrix} [\,7 \quad 8\,].$$

Although equivalent mathematically, it turns out that these versions of matrix multiplication can have very different levels of performance because of their memory traffic properties. This matter is pursued in §1.4. For now, it is worth detailing the above three approaches to matrix multiplication because it gives us a chance to review notation and to practice thinking at different linear algebraic levels.

1.1.11 Scalar-Level Specifications

To fix the discussion we focus on the following matrix multiplication update:

$$C = AB + C \qquad A \in \mathbb{R}^{m\times p},\ B \in \mathbb{R}^{p\times n},\ C \in \mathbb{R}^{m\times n}.$$

The starting point is the familiar triply-nested loop algorithm:

Algorithm 1.1.5 (Matrix Multiplication: ijk Variant) If $A \in \mathbb{R}^{m\times p}$, $B \in \mathbb{R}^{p\times n}$, and $C \in \mathbb{R}^{m\times n}$ are given, then this algorithm overwrites C with $AB + C$.

```
for i = 1:m
    for j = 1:n
        for k = 1:p
            C(i,j) = A(i,k)B(k,j) + C(i,j)
        end
    end
end
```


This is the "ijk variant" because we identify the rows of C (and A) with i, the columns of C (and B) with j, and the summation index with k.

We consider the update $C = AB + C$ instead of just $C = AB$ for two reasons. We do not have to bother with $C = 0$ initializations and updates of the form $C = AB + C$ arise more frequently in practice.

The three loops in the matrix multiplication update can be arbitrarily ordered giving $3! = 6$ variations. Thus,

```
for j = 1:n
    for k = 1:p
        for i = 1:m
            C(i,j) = A(i,k)B(k,j) + C(i,j)
        end
    end
end
```

is the jki variant. Each of the six possibilities (ijk, jik, ikj, jki, kij, kji) features an inner loop operation (dot product or saxpy) and has its own pattern of data flow. For example, in the ijk variant, the inner loop oversees a dot product that requires access to a row of A and a column of B. The jki variant involves a saxpy that requires access to a column of C and a column of A. These attributes are summarized in Table 1.1.1 along with an interpretation of what is going on when the middle and inner loop are considered together. Each variant involves the same amount of floating point arithmetic, but accesses the A, B, and C data differently.

Loop Order	Inner Loop	Middle Loop	Inner Loop Data Access
ijk	dot	vector × matrix	A by row, B by column
jik	dot	matrix × vector	A by row, B by column
ikj	saxpy	row gaxpy	B by row, C by row
jki	saxpy	column gaxpy	A by column, C by column
kij	saxpy	row outer product	B by row, C by row
kji	saxpy	column outer product	A by column, C by column

TABLE 1.1.1. *Matrix Multiplication: Loop Orderings and Properties*

1.1.12 A Dot Product Formulation

The usual matrix multiplication procedure regards AB as an array of dot products to be computed one at a time in left-to-right, top-to-bottom order.

This is the idea behind Algorithm 1.1.5. Using the colon notation we can highlight this dot-product formulation:

Algorithm 1.1.6 (Matrix Multiplication: Dot Product Version) If $A \in \mathbb{R}^{m \times p}$, $B \in \mathbb{R}^{p \times n}$, and $C \in \mathbb{R}^{m \times n}$ are given, then this algorithm overwrites C with $AB + C$.

```
for i = 1:m
    for j = 1:n
        C(i,j) = A(i,:)B(:,j) + C(i,j)
    end
end
```

In the language of partitioned matrices, if

$$A = \begin{bmatrix} a_1^T \\ \vdots \\ a_m^T \end{bmatrix} \qquad a_k \in \mathbb{R}^p$$

and

$$B = [\, b_1, \ldots, b_n \,] \qquad b_k \in \mathbb{R}^p$$

then Algorithm 1.1.6 has this interpretation:

```
for i = 1:m
    for j = 1:n
        c_ij = a_i^T b_j + c_ij
    end
end
```

Note that the "mission" of the j-loop is to compute the ith row of the update. To emphasize this we could write

```
for i = 1:m
    c_i^T = a_i^T B + c_i^T
end
```

where

$$C = \begin{bmatrix} c_1^T \\ \vdots \\ c_m^T \end{bmatrix}$$

is a row partitioning of C. To say the same thing with the colon notation we write

```
for i = 1:m
    C(i,:) = A(i,:)B + C(i,:)
end
```

Either way we see that the inner two loops of the ijk variant define a row-oriented gaxpy operation.

1.1.13 A Saxpy Formulation

Suppose A and C are column-partitioned as follows

$$A = [\, a_1, \ldots, a_p \,] \qquad a_j \in \mathbb{R}^m$$

$$C = [\, c_1, \ldots, c_n \,] \qquad c_j \in \mathbb{R}^m.$$

By comparing jth columns in $C = AB + C$ we see that

$$c_j = \sum_{k=1}^{p} b_{kj} a_k + c_j, \qquad j = 1{:}n.$$

These vector sums can be put together with a sequence of saxpy updates.

Algorithm 1.1.7 (Matrix Multiplication: Saxpy Version) If the matrices $A \in \mathbb{R}^{m \times p}$, $B \in \mathbb{R}^{p \times n}$, and $C \in \mathbb{R}^{m \times n}$ are given, then this algorithm overwrites C with $AB + C$.

```
for j = 1:n
    for k = 1:p
        C(:,j) = A(:,k)B(k,j) + C(:,j)
    end
end
```

Note that the k-loop oversees a gaxpy operation:

```
for j = 1:n
    C(:,j) = AB(:,j) + C(:,j)
end
```

1.1.14 An Outer Product Formulation

Consider the kij variant of Algorithm 1.1.5:

```
for k = 1:p
    for j = 1:n
        for i = 1:m
            C(i,j) = A(i,k)B(k,j) + C(i,j)
        end
    end
end
```

The inner two loops oversee the outer product update

$$C = a_k b_k^T + C$$

where

$$A = [\, a_1, \ldots, a_p \,] \quad \text{and} \quad B = \begin{bmatrix} b_1^T \\ \vdots \\ b_p^T \end{bmatrix} \qquad (1.1.3)$$

with $a_k \in \mathbb{R}^m$ and $b_k \in \mathbb{R}^n$. We therefore obtain

Algorithm 1.1.8 (Matrix Multiplication: Outer Product Version) If $A \in \mathbb{R}^{m\times p}$, $B \in \mathbb{R}^{p\times n}$, and $C \in \mathbb{R}^{m\times n}$ are given, then this algorithm overwrites C with $AB + C$.

```
for k = 1:p
      C = A(:,k)B(k,:) + C
end
```

This implementation revolves around the fact that AB is the sum of p outer products.

1.1.15 The Notion of "Level"

The dot product and saxpy operations are examples of "level-1" operations. Level-1 operations involve an amount of data and an amount of arithmetic that is linear in the dimension of the operation. An m-by-n outer product update or gaxpy operation involves a quadratic amount of data ($O(mn)$) and a quadratic amount of work ($O(mn)$). They are examples of "level-2" operations.

The matrix update $C = AB + C$ is a "level-3" operation. Level-3 operations involve a quadratic amount of data and a cubic amount of work. If A, B, and C are n-by-n matrices, then $C = AB + C$ involves $O(n^2)$ matrix entries and $O(n^3)$ arithmetic operations.

The design of matrix algorithms that are rich in high-level linear algebra operations is a recurring theme in the book. For example, a high performance linear equation solver may require a level-3 organization of Gaussian elimination. This requires some algorithmic rethinking because that method is usually specified in level-1 terms, e.g., "multiply row 1 by a scalar and add the result to row 2."

1.1.16 A Note on Matrix Equations

In striving to understand matrix multiplication via outer products, we essentially established the matrix equation

$$AB = \sum_{k=1}^{p} a_k b_k^T$$

where the a_k and b_k are defined by the partitionings in (1.1.3).

Numerous matrix equations are developed in subsequent chapters. Sometimes they are established algorithmically like the above outer product expansion and other times they are proved at the ij-component level. As an example of the latter, we prove an important result that characterizes transposes of products.

Theorem 1.1.1 *If $A \in \mathbb{R}^{m\times p}$ and $B \in \mathbb{R}^{p\times n}$, then $(AB)^T = B^TA^T$.*

Proof. If $C = (AB)^T$, then

$$c_{ij} = [(AB)^T]_{ij} = [AB]_{ji} = \sum_{k=1}^{p} a_{jk}b_{ki}\,.$$

On the other hand, if $D = B^TA^T$, then

$$d_{ij} = [B^TA^T]_{ij} = \sum_{k=1}^{p}[B^T]_{ik}[A^T]_{kj} = \sum_{k=1}^{p} b_{ki}a_{jk}.$$

Since $c_{ij} = d_{ij}$ for all i and j, it follows that $C = D$. □

Scalar-level proofs such as this one are usually not very insightful. However, they are sometimes the only way to proceed.

1.1.17 Complex Matrices

From time to time computations that involve complex matrices are discussed. The vector space of m-by-n complex matrices is designated by $\mathbb{C}^{m\times n}$. The scaling, addition, and multiplication of complex matrices corresponds exactly to the real case. However, transposition becomes *conjugate transposition*:

$$C = A^H \quad \Longrightarrow \quad c_{ij} = \bar{a}_{ji}\,.$$

The vector space of complex n-vectors is designated by $\mathbb{C}^n$. The dot product of complex n-vectors x and y is prescribed by

$$s = x^Hy = \sum_{i=1}^{n} \bar{x}_i y_i\,.$$

Finally, if $A = B + iC \in \mathbb{C}^{m\times n}$, then we designate the real and imaginary parts of A by $\text{Re}(A) = B$ and $\text{Im}(A) = C$ respectively.

Problems

P1.1.1 Suppose $A \in \mathbb{R}^{n\times n}$ and $x \in \mathbb{R}^r$ are given. Give a saxpy algorithm for computing the first column of $M = (A - x_1I)\cdots(A - x_rI)$.

P1.1.2 In the conventional 2-by-2 matrix multiplication $C = AB$, there are eight multiplications: $a_{11}b_{11}$, $a_{11}b_{12}$, $a_{21}b_{11}$, $a_{21}b_{12}$, $a_{12}b_{21}$, $a_{12}b_{22}$, $a_{22}b_{21}$ and $a_{22}b_{22}$. Make a table that indicates the order that these multiplications are performed for the ijk, jik, kij, ikj, jki, and kji matrix multiply algorithms.

P1.1.3 Give an algorithm for computing $C = (xy^T)^k$ where x and y are n-vectors.

P1.1.4 Specify an algorithm for computing $(XY^T)^k$ where $X, Y \in \mathbb{R}^{n\times 2}$.

P1.1.5 Formulate an outer product algorithm for the update $C = AB^T + C$ where $A \in \mathbb{R}^{m\times r}$, $B \in \mathbb{R}^{n\times r}$, and $C \in \mathbb{R}^{m\times n}$.

P1.1.6 Suppose we have real n-by-n matrices C, D, E, and F. Show how to compute real n-by-n matrices A and B with just three real n-by-n matrix multiplications so that $(A + iB) = (C + iD)(E + iF)$. Hint: Compute $W = (C + D)(E - F)$.

Notes and References for Sec. 1.1

It must be stressed that the development of quality software from any of our "semi-formal" algorithmic presentations is a long and arduous task. Even the implementation of the level-1,2, and 3 BLAS require care:

C.L. Lawson, R.J. Hanson, , D.R. Kincaid, and F.T. Krogh (1979). "Basic Linear Algebra Subprograms for FORTRAN Usage," *ACM Trans. Math. Soft. 5*, 308–323.

C.L. Lawson, R.J. Hanson, D.R. Kincaid, and F.T. Krogh (1979). "Algorithm 539, Basic Linear Algebra Subprograms for FORTRAN Usage," *ACM Trans. Math. Soft. 5*, 324–325.

J.J. Dongarra, J. Du Croz, S. Hammarling, and R.J. Hanson (1988). "An Extended Set of Fortran Basic Linear Algebra Subprograms," *ACM Trans. Math. Soft. 14*, 1–17.

J.J. Dongarra, J. Du Croz, S. Hammarling, and R.J. Hanson (1988). "Algorithm 656 An Extended Set of Fortran Basic Linear Algebra Subprograms: Model Implementation and Test Programs," *ACM Trans. Math. Soft. 14*, 18–32.

J.J. Dongarra, J. Du Croz, I.S. Duff, and S.J. Hammarling (1990). "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. Math. Soft. 16*, 1–17.

J.J. Dongarra, J. Du Croz, I.S. Duff, and S.J. Hammarling (1990). "Algorithm 679. A Set of Level 3 Basic Linear Algebra Subprograms: Model Implementation and Test Programs," *ACM Trans. Math. Soft. 16*, 18–28.

Other BLAS references include

B. Kågström, P. Ling, and C. Van Loan (1991). "High-Performance Level-3 BLAS: Sample Routines for Double Precision Real Data," in *High Performance Computing II*, M. Durand and F. El Dabaghi (eds), North-Holland, 269–281.

B. Kågström, P. Ling, and C. Van Loan (1995). "GEMM-Based Level-3 BLAS: High-Performance Model Implementations and Performance Evaluation Benchmark," in *Parallel Programming and Applications*, P. Fritzon and L. Finmo (eds), ISO Press, 184–188.

For an appreciation of the subtleties associated with software development we recommend

J.R. Rice (1981). *Matrix Computations and Mathematical Software*, Academic Press, New York.

and a browse through the LAPACK manual.

1.2 Exploiting Structure

The efficiency of a given matrix algorithm depends on many things. Most obvious and what we treat in this section is the amount of required arithmetic and storage. We continue to use matrix-vector and matrix-matrix multiplication as a vehicle for introducing the key ideas. As examples of exploitable structure we have chosen the properties of bandedness and symmetry. Band matrices have many zero entries and so it is no surprise that band matrix manipulation allows for many arithmetic and storage shortcuts. Arithmetic complexity and data structures are discussed in this context.

Symmetric matrices provide another set of examples that can be used to illustrate structure exploitation. Symmetric linear systems and eigenvalue problems have a very prominent role to play in matrix computations and so it is important to be familiar with their manipulation.

1.2.1 Band Matrices and the x-0 Notation

We say that $A \in \mathbb{R}^{m\times n}$ has *lower bandwidth* p if $a_{ij} = 0$ whenever $i > j + p$ and *upper bandwidth* q if $j > i + q$ implies $a_{ij} = 0$. Here is an example of an 8-by-5 matrix that has lower bandwidth 1 and upper bandwidth 2:

$$\begin{bmatrix} \times & \times & \times & 0 & 0 \\ \times & \times & \times & \times & 0 \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & \times & \times \\ 0 & 0 & 0 & 0 & \times \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

The $\times$'s designates arbitrary nonzero entries. This notation is handy to indicate the zero-nonzero structure of a matrix and we use it extensively. Band structures that occur frequently are tabulated in Table 1.2.1.

1.2.2 Diagonal Matrix Manipulation

Matrices with upper and lower bandwidth zero are *diagonal*. If $D \in \mathbb{R}^{m\times n}$ is diagonal, then

$$D = \text{diag}(d_1, \ldots, d_q), \quad q = \min\{m, n\} \quad \Longleftrightarrow \quad d_i = d_{ii}$$

If D is diagonal and A is a matrix, then DA is a *row scaling* of A and AD is a *column scaling* of A.

Type of Matrix	Lower Bandwidth	Upper Bandwidth
diagonal	0	0
upper triangular	0	$n-1$
lower triangular	$m-1$	0
tridiagonal	1	1
upper bidiagonal	0	1
lower bidiagonal	1	0
upper Hessenberg	1	$n-1$
lower Hessenberg	$m-1$	1

TABLE 1.2.1. *Band Terminology for m-by-n Matrices*

1.2.3 Triangular Matrix Multiplication

To introduce band matrix "thinking" we look at the matrix multiplication problem $C = AB$ when A and B are both n-by-n and upper triangular. The 3-by-3 case is illuminating:

$$C = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} + a_{12}b_{22} & a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} \\ 0 & a_{22}b_{22} & a_{22}b_{23} + a_{23}b_{33} \\ 0 & 0 & a_{33}b_{33} \end{bmatrix}.$$

It suggests that the product is upper triangular and that its upper triangular entries are the result of abbreviated inner products. Indeed, since $a_{ik}b_{kj} = 0$ whenever $k < i$ or $j < k$ we see that

$$c_{ij} = \sum_{k=i}^{j} a_{ik}b_{kj}$$

and so we obtain:

Algorithm 1.2.1 (Triangular Matrix Multiplication) If $A, B \in \mathbb{R}^{n\times n}$ are upper triangular, then this algorithm computes $C = AB$.

```
C = 0
for i = 1:n
    for j = i:n
        for k = i:j
            C(i,j) = A(i,k)B(k,j) + C(i,j)
        end
    end
end
```


To quantify the savings in this algorithm we need some tools for measuring the amount of work.

1.2.4 Flops

Obviously, upper triangular matrix multiplication involves less arithmetic than when the matrices are full. One way to quantify this is with the notion of a *flop*. A flop[1] is a floating point operation. A dot product or saxpy operation of length n involves $2n$ flops because there are n multiplications and n adds in either of these vector operations.

The gaxpy $y = Ax + y$ where $A \in \mathbb{R}^{m\times n}$ involves $2mn$ flops as does an m-by-n outer product update of the form $A = A + xy^T$.

The matrix multiply update $C = AB + C$ where $A \in \mathbb{R}^{m\times p}$, $B \in \mathbb{R}^{p\times n}$, and $C \in \mathbb{R}^{m\times n}$ involves $2mnp$ flops.

Flop counts are usually obtained by summing the amount of arithmetic associated with the most deeply nested statements in an algorithm. For matrix-matrix multiplication, this is the statement,

$$C(i,j) = A(i,k)B(k,j) + C(i,j)$$

which involves two flops and is executed mnp times as a simple loop accounting indicates. Hence the conclusion that general matrix multiplication requires $2mnp$ flops.

Now let us investigate the amount of work involved in Algorithm 1.2.1. Note that c_{ij}, $(i \le j)$ requires $2(j - i + 1)$ flops. Using the heuristics

$$\sum_{p=1}^{q} p = \frac{q(q+1)}{2} \approx \frac{q^2}{2}$$

and

$$\sum_{p=1}^{q} p^2 = \frac{q^3}{3} + \frac{q^2}{2} + \frac{q}{6} \approx \frac{q^3}{3}$$

we find that triangular matrix multiplication requires one-sixth the number of flops as full matrix multiplication:

$$\sum_{i=1}^{n}\sum_{j=i}^{n} 2(j-i+1) = \sum_{i=1}^{n}\sum_{j=1}^{n-i+1} 2j \approx \sum_{i=1}^{n} \frac{2(n-i+1)^2}{2} = \sum_{i=1}^{n} i^2 \approx \frac{n^3}{3}.$$

We throw away the low order terms since their inclusion does not contribute to what the flop count "says." For example, an exact flop count of Algorithm 1.2.1 reveals that precisely $n^3/3 + n^2 + 2n/3$ flops are involved. For

[1] In the first edition of this book we defined a flop to be the amount of work associated with an operation of the form $a_{ij} = a_{ij} + a_{ik}a_{kj}$, i.e., a floating point add, a floating point multiply, and some subscripting. Thus, an "old flop" involves two "new flops." In defining a flop to be a single floating point operation we are opting for a more precise measure of arithmetic complexity.

large n (the typical situation of interest) we see that the exact flop count offers no insight beyond the $n^3/3$ approximation.

Flop counting is a necessarily crude approach to the measuring of program efficiency since it ignores subscripting, memory traffic, and the countless other overheads associated with program execution. We must not infer too much from a comparison of flops counts. We cannot conclude, for example, that triangular matrix multiplication is six times faster than square matrix multiplication. Flop counting is just a "quick and dirty" accounting method that captures only one of the several dimensions of the efficiency issue.

1.2.5 The Colon Notation–Again

The dot product that the k-loop performs in Algorithm 1.2.1 can be succinctly stated if we extend the colon notation introduced in §1.1.8. Suppose $A \in \mathbb{R}^{m\times n}$ and the integers p, q, and r satisfy $1 \le p \le q \le n$ and $1 \le r \le m$. We then define

$$A(r, p{:}q) = [\, a_{rp}, \ldots, a_{rq} \,] \in \mathbb{R}^{1\times(q-p+1)}.$$

Likewise, if $1 \le p \le q \le m$ and $1 \le c \le n$, then

$$A(p{:}q, c) = \begin{bmatrix} a_{pc} \\ \vdots \\ a_{qc} \end{bmatrix} \in \mathbb{R}^{q-p+1}.$$

With this notation we can rewrite Algorithm 1.2.1 as

$C(1{:}n, 1{:}n) = 0$
for $i = 1{:}n$
  for $j = i{:}n$
    $C(i,j) = A(i, i{:}j)B(i{:}j, j) + C(i,j)$
  end
end

We mention one additional feature of the colon notation. Negative increments are allowed. Thus, if x and y are n-vectors, then $s = x^T y(n{:}-1{:}1)$ is the summation

$$s = \sum_{i=1}^{n} x_i y_{n-i+1}.$$

1.2.6 Band Storage

Suppose $A \in \mathbb{R}^{n\times n}$ has lower bandwidth p and upper bandwidth q and assume that p and q are much smaller than n. Such a matrix can be stored in a $(p+q+1)$-by-n array $A.band$ with the convention that

$$a_{ij} = A.band(i - j + q + 1, j) \qquad (1.2.1)$$

for all (i,j) that fall inside the band. Thus, if

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & a_{35} & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} & a_{46} \\ 0 & 0 & 0 & a_{54} & a_{55} & a_{56} \\ 0 & 0 & 0 & 0 & a_{65} & a_{66} \end{bmatrix},$$

then

$$A.band = \begin{bmatrix} 0 & 0 & a_{13} & a_{24} & a_{35} & a_{46} \\ 0 & a_{12} & a_{23} & a_{34} & a_{45} & a_{56} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} \\ a_{21} & a_{32} & a_{43} & a_{54} & a_{65} & 0 \end{bmatrix}.$$

Here, the "0" entries are unused. With this data structure, our column-oriented gaxpy algorithm transforms to the following:

Algorithm 1.2.2 (Band Gaxpy) Suppose $A \in \mathbb{R}^{n\times n}$ has lower bandwidth p and upper bandwidth q and is stored in the $A.band$ format (1.2.1). If $x, y \in \mathbb{R}^n$, then this algorithm overwrites y with $Ax + y$.

for $j = 1{:}n$
  $y_{top} = \max(1, j - q)$
  $y_{bot} = \min(n, j + p)$
  $a_{top} = \max(1, q + 2 - j)$
  $a_{bot} = a_{top} + y_{bot} - y_{top}$
  $y(y_{top}{:}y_{bot}) = x(j)A.band(a_{top}{:}a_{bot}, j) + y(y_{top}{:}y_{bot})$
end

Notice that by storing A by column in $A.band$, we obtain a saxpy, column access procedure. Indeed, Algorithm 1.2.2 is obtained from Algorithm 1.1.4 by recognizing that each saxpy involves a vector with a small number of nonzeros. Integer arithmetic is used to identify the location of these nonzeros. As a result of this careful zero/nonzero analysis, the algorithm involves just $2n(p+q+1)$ flops with the assumption that p and q are much smaller than n.

1.2.7 Symmetry

We say that $A \in \mathbb{R}^{n\times n}$ is *symmetric* if $A^T = A$. Thus,

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{bmatrix}$$

is symmetric. Storage requirements can be halved if we just store the lower triangle of elements, e.g., $A.vec = [\ 1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6\ ]$. In general, with this data structure we agree to store the a_{ij} as follows:

$$a_{ij} = A.vec((j-1)n - j(j-1)/2 + i) \qquad (i \geq j) \tag{1.2.2}$$

Let us look at the column-oriented gaxpy operation with the matrix A represented in $A.vec$.

Algorithm 1.2.3 (Symmetric Storage Gaxpy) Suppose $A \in \mathbb{R}^{n\times n}$ is symmetric and stored in the $A.vec$ style (1.2.2). If $x, y \in \mathbb{R}^n$, then this algorithm overwrites y with $Ax + y$.

```
for j = 1:n
    for i = 1:j - 1
        y(i) = A.vec((i - 1)n - i(i - 1)/2 + j)x(j) + y(i)
    end
    for i = j:n
        y(i) = A.vec((j - 1)n - j(j - 1)/2 + i)x(j) + y(i)
    end
end
```

This algorithm requires the same $2n^2$ flops that an ordinary gaxpy requires. Notice that the halving of the storage requirement is purchased with some awkward subscripting.

1.2.8 Store by Diagonal

Symmetric matrices can also be stored by diagonal. If

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{bmatrix},$$

then in a store-by-diagonal scheme we represent A with the vector

$$A.diag = [\ 1\ \ 4\ \ 6\ \ 2\ \ 5\ \ 3\ ]\ .$$

In general, if $i \geq j$, then

$$a_{i+k,i} = A.diag(i + nk - k(k-1)/2) \qquad (k \geq 0) \tag{1.2.3}$$

Some notation simplifies the discussion of how to use this data structure in a matrix-vector multiplication.

If $A \in \mathbb{R}^{m\times n}$, then let $D(A,k) \in \mathbb{R}^{m\times n}$ designate the kth diagonal of A as follows:

$$[D(A,k)]_{ij} = \begin{cases} a_{ij} & j = i + k, \quad 1 \leq i \leq m, \quad 1 \leq j \leq n \\ 0 & \text{otherwise.} \end{cases}$$

Thus,

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & 0 & 3 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}}_{D(A,2)} + \underbrace{\begin{bmatrix} 0 & 2 & 0 \\ 0 & 0 & 5 \\ 0 & 0 & 0 \end{bmatrix}}_{D(A,1)}$$

$$+ \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 6 \end{bmatrix}}_{D(A,0)} + \underbrace{\begin{bmatrix} 0 & 0 & 0 \\ 2 & 0 & 0 \\ 0 & 5 & 0 \end{bmatrix}}_{D(A,-1)} + \underbrace{\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 3 & 0 & 0 \end{bmatrix}}_{D(A,-2)}.$$

Returning to our store-by-diagonal data structure, we see that the nonzero parts of $D(A,0)$, $D(A,1), \ldots, D(A,n-1)$ are sequentially stored in the $A.diag$ scheme (1.2.3). The gaxpy $y = Ax + y$ can then be organized as follows:

$$y = D(A,0)x + \sum_{k=1}^{n-1}(D(A,k) + D(A,k)^T)x + y\,.$$

Working out the details we obtain the following algorithm.

Algorithm 1.2.4 (Store-By-Diagonal Gaxpy) Suppose $A \in \mathbb{R}^{n\times n}$ is symmetric and stored in the $A.diag$ style (1.2.3). If $x, y \in \mathbb{R}^n$, then this algorithm overwrites y with $Ax + y$.

```
for i = 1:n
    y(i) = A.diag(i)x(i) + y(i)
end
for k = 1:n - 1
    t = nk - k(k - 1)/2
    {y = D(A,k)x + y}
    for i = 1:n - k
        y(i) = A.diag(i + t)x(i + k) + y(i)
    end
    {y = D(A,k)^T x + y}
    for i = 1:n - k
        y(i + k) = A.diag(i + t)x(i) + y(i + k)
    end
end
```

Note that the inner loops oversee vector multiplications:

$$y(1{:}n-k) = A.diag(t+1{:}t+n-k).*x(k+1{:}n) + y(1{:}n-k)$$
$$y(k+1{:}n) = A.diag(t+1{:}t+n-k).*x(1{:}n-k) + y(k+1{:}n)$$

1.2.9 A Note on Overwriting and Workspaces

An undercurrent in the above discussion has been the economical use of storage. Overwriting input data is another way to control the amount of memory that a matrix computation requires. Consider the n-by-n matrix multiplication problem $C = AB$ with the proviso that the "input matrix" B is to be overwritten by the "output matrix" C. We cannot simply transform

```
C(1:n, 1:n) = 0
for j = 1:n
    for k = 1:n
        C(:, j) = C(:, j) + A(:, k)B(k, j)
    end
end
```

to

```
for j = 1:n
    for k = 1:n
        B(:, j) = B(:, j) + A(:, k)B(k, j)
    end
end
```

because $B(:, j)$ is needed throughout the entire k-loop. A linear *workspace* is needed to hold the jth column of the product until it is "safe" to overwrite $B(:, j)$:

```
for j = 1:n
    w(1:n) = 0
    for k = 1:n
        w(:) = w(:) + A(:, k)B(k, j)
    end
    B(:, j) = w(:)
end
```

A linear workspace overhead is usually not important in a matrix computation that has a 2-dimensional array of the same order.

Problems

P1.2.1 Give an algorithm that overwrites A with A^2 where $A \in \mathbb{R}^{n\times n}$ is (a) upper triangular and (b) square. Strive for a minimum workspace in each case.

P1.2.2 Suppose $A \in \mathbb{R}^{n\times n}$ is upper Hessenberg and that scalars $\lambda_1, \ldots, \lambda_r$ are given. Give a saxpy algorithm for computing the first column of $M = (A - \lambda_1 I) \cdots (A - \lambda_r I)$.

P1.2.3 Give a column saxpy algorithm for the n-by-n matrix multiplication problem $C = AB$ where A is upper triangular and B is lower triangular.

P1.2.4 Extend Algorithm 1.2.2 so that it can handle rectangular band matrices. Be sure to describe the underlying data structure.

P1.2.5 $A \in \mathbb{R}^{n\times n}$ is *Hermitian* if $A^H = A$. If $A = B + iC$, then it is easy to show that $B^T = B$ and $C^T = -C$. Suppose we represent A in an array $A.herm$ with the property that $A.herm(i, j)$ houses b_{ij} if $i \geq j$ and c_{ij} if $j > i$. Using this data structure write a matrix-vector multiply function that computes Re(z) and Im(z) from Re(x) and Im(x) so that $z = Ax$.

P1.2.6 Suppose $X \in \mathbb{R}^{n\times p}$ and $A \in \mathbb{R}^{n\times n}$, with A symmetric and stored by diagonal. Give an algorithm that computes $Y = X^TAX$ and stores the result by diagonal. Use separate arrays for A and Y.

P1.2.7 Suppose $a \in \mathbb{R}^n$ is given and that $A \in \mathbb{R}^{n\times n}$ has the property that $a_{ij} = a_{|i-j|+1}$. Give an algorithm that overwrites y with $Ax + y$ where $x, y \in \mathbb{R}^n$ are given.

P1.2.8 Suppose $a \in \mathbb{R}^n$ is given and that $A \in \mathbb{R}^{n\times n}$ has the property that $a_{ij} = a_{((i+j-1) \bmod n)+1}$. Give an algorithm that overwrites y with $Ax + y$ where $x, y \in \mathbb{R}^n$ are given.

P1.2.9 Develop a compact store-by-diagonal scheme for unsymmetric band matrices and write the corresponding gaxpy algorithm.

P1.2.10 Suppose p and q are n-vectors and that $A = (a_{ij})$ is defined by $a_{ij} = a_{ji} = p_iq_j$ for $1 \leq i \leq j \leq n$. How many flops are required to compute $y = Ax$ where $x \in \mathbb{R}^n$ is given?

Notes and References for Sec. 1.2

Consult the LAPACK manual for a discussion about appropriate data structures when symmetry and/or bandedness is present. See also

N. Madsen, G. Roderigue, and J. Karush (1976). "Matrix Multiplication by Diagonals on a Vector Parallel Processor," *Infomation Processing Letters 5*, 41-45.

1.3 Block Matrices and Algorithms

Having a facility with block matrix notation is crucial in matrix computations because it simplifies the derivation of many central algorithms. Moreover, "block algorithms" are increasingly important in high performance computing. By a block algorithm we essentially mean an algorithm that is rich in matrix-matrix multiplication. Algorithms of this type turn out to be more efficient in many computing environments than those that are organized at a lower linear algebraic level.

1.3.1 Block Matrix Notation

Column and row partitionings are special cases of matrix blocking. In general we can partition both the rows and columns of an m-by-n matrix

A to obtain

$$A = \begin{bmatrix} A_{11} & \dots & A_{1r} \\ \vdots & & \vdots \\ A_{q1} & \cdots & A_{qr} \end{bmatrix} \begin{matrix} m_1 \\ \\ m_q \end{matrix}$$
$$\quad n_1 \qquad n_r$$

where $m_1 + \cdots + m_q = m$, $n_1 + \cdots + n_r = n$, and $A_{\alpha\beta}$ designates the (α, β) block or submatrix. With this notation, block $A_{\alpha\beta}$ has dimension m_α-by-n_β and we say that $A = (A_{\alpha\beta})$ is a q-by-r block matrix.

1.3.2 Block Matrix Manipulation

Block matrices combine just like matrices with scalar entries as long as certain dimension requirements are met. For example, if

$$B = \begin{bmatrix} B_{11} & \dots & B_{1r} \\ \vdots & & \vdots \\ B_{q1} & \cdots & B_{qr} \end{bmatrix} \begin{matrix} m_1 \\ \\ m_q \end{matrix} ,$$
$$\quad n_1 \qquad n_r$$

then we say that B is partitioned *conformably* with the matrix A above. The sum $C = A + B$ can also be regarded as a q-by-r block matrix:

$$C = \begin{bmatrix} C_{11} & \cdots & C_{1r} \\ \vdots & & \vdots \\ C_{q1} & \cdots & C_{qr} \end{bmatrix} = \begin{bmatrix} A_{11} + B_{11} & \cdots & A_{1r} + B_{1r} \\ \vdots & & \vdots \\ A_{q1} + B_{q1} & \cdots & A_{qr} + B_{qr} \end{bmatrix}.$$

The multiplication of block matrices is a little trickier. We start with a pair of lemmas.

Lemma 1.3.1 *If $A \in \mathbb{R}^{m\times p}$, $B \in \mathbb{R}^{p\times n}$,*

$$A = \begin{bmatrix} A_1 \\ \vdots \\ A_q \end{bmatrix} \begin{matrix} m_1 \\ \\ m_q \end{matrix} \qquad B = \begin{bmatrix} B_1 & ,\dots, & B_r \end{bmatrix} ,$$
$$\qquad\qquad\qquad\qquad n_1 \qquad n_r$$

then

$$AB = C = \begin{bmatrix} C_{11} & \dots & C_{1r} \\ \vdots & & \vdots \\ C_{q1} & \cdots & C_{qr} \end{bmatrix} \begin{matrix} m_1 \\ \\ m_q \end{matrix}$$
$$\quad n_1 \qquad n_r$$

where $C_{\alpha\beta} = A_\alpha B_\beta$ for $\alpha = 1{:}q$ and $\beta = 1{:}r$.

Proof. First we relate scalar entries in block $C_{\alpha\beta}$ to scalar entries in C. For $1 \le \alpha \le q$, $1 \le \beta \le r$, $1 \le i \le m_\alpha$, and $1 \le j \le n_\beta$ we have

$$[C_{\alpha\beta}]_{ij} = c_{\lambda+i,\mu+j}$$

where

$$\begin{aligned} \lambda &= m_1 + \cdots + m_{\alpha-1} \\ \mu &= n_1 + \cdots + n_{\beta-1}. \end{aligned}$$

But

$$c_{\lambda+i,\mu+j} = \sum_{k=1}^{p} a_{\lambda+i,k} b_{k,\mu+j} = \sum_{k=1}^{p} [A_\alpha]_{ik} [B_\beta]_{kj} = [A_\alpha B_\beta]_{ij}.$$

Thus, $C_{\alpha\beta} = A_\alpha B_\beta$. □

Lemma 1.3.2 *If $A \in \mathbb{R}^{m\times p}$, $B \in \mathbb{R}^{p\times n}$,*

$$A = \begin{bmatrix} A_1 & ,\dots, & A_s \end{bmatrix} \quad , \textit{and} \quad B = \begin{bmatrix} B_1 \\ \vdots \\ B_s \end{bmatrix} \begin{matrix} p_1 \\ \\ p_s \end{matrix} ,$$
$$\quad p_1 \qquad p_s$$

then

$$AB = C = \sum_{\gamma=1}^{s} A_\gamma B_\gamma.$$

Proof. We set $s = 2$ and leave the general s case to the reader. (See P1.3.6.) For $1 \le i \le m$ and $1 \le j \le n$ we have

$$\begin{aligned} c_{ij} &= \sum_{k=1}^{p} a_{ik} b_{kj} = \sum_{k=1}^{p_1} a_{ik} b_{kj} + \sum_{k=p_1+1}^{p_1+p_2} a_{ik} b_{kj} \\ &= [A_1 B_1]_{ij} + [A_2 B_2]_{ij} = [A_1 B_1 + A_2 B_2]_{ij}. \end{aligned}$$

Thus, $C = A_1 B_1 + A_2 B_2$. □

For general block matrix multiplication we have the following result:

Theorem 1.3.3 *If*

$$A = \begin{bmatrix} A_{11} & \dots & A_{1s} \\ \vdots & & \vdots \\ A_{q1} & \cdots & A_{qs} \end{bmatrix} \begin{matrix} m_1 \\ \\ m_q \end{matrix} , \qquad B = \begin{bmatrix} B_{11} & \dots & B_{1r} \\ \vdots & & \vdots \\ B_{s1} & \cdots & B_{sr} \end{bmatrix} \begin{matrix} p_1 \\ \\ p_s \end{matrix} ,$$
$$\quad p_1 \qquad p_s \qquad\qquad\qquad n_1 \qquad n_r$$

and we partition the product $C = AB$ as follows,

$$C = \begin{bmatrix} C_{11} & \dots & C_{1r} \\ \vdots & & \vdots \\ C_{q1} & \cdots & C_{qr} \end{bmatrix} \begin{matrix} m_1 \\ \\ m_q \end{matrix} , \qquad \begin{matrix} n_1 & & n_r \end{matrix}$$

then

$$C_{\alpha\beta} = \sum_{\gamma=1}^{s} A_{\alpha\gamma} B_{\gamma\beta} \qquad \alpha = 1{:}q, \quad \beta = 1{:}r.$$

Proof. See P1.3.7. □

A very important special case arises if we set $s = 2$, $r = 1$, and $n_1 = 1$:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} A_{11}x_1 + A_{12}x_2 \\ A_{21}x_1 + A_{22}x_2 \end{bmatrix}.$$

This partitioned matrix-vector product is used over and over again in subsequent chapters.

1.3.3 Submatrix Designation

As with "ordinary" matrix multiplication, block matrix multiplication can be organized in several ways. To specify the computations precisely, we need some notation.

Suppose $A \in \mathbb{R}^{m \times n}$ and that $i = (i_1, \dots, i_r)$ and $j = (j_1, \dots, j_c)$ are integer vectors with the property that

$$\begin{aligned} i_1, \dots, i_r &\in \{1, 2, \dots, m\} \\ j_1, \dots, j_c &\in \{1, 2, \dots, n\}. \end{aligned}$$

We let $A(i, j)$ denote the r-by-c submatrix

$$A(i,j) = \begin{bmatrix} A(i_1, j_1) & \cdots & A(i_1, j_c) \\ \vdots & & \vdots \\ A(i_r, j_1) & \cdots & A(i_r, j_c) \end{bmatrix}.$$

If the entries in the subscript vectors i and j are contiguous, then the "colon" notation can be used to define $A(i, j)$ in terms of the scalar entries in A. In particular, if $1 \le i_1 \le i_2 \le m$ and $1 \le j_1 \le j_2 \le n$, then $A(i_1{:}i_2, j_1{:}j_2)$ is the submatrix obtained by extracting rows i_1 through i_2 and columns j_1 through j_2, e.g,

$$A(3{:}5, 1{:}2) = \begin{bmatrix} a_{31} & a_{32} \\ a_{41} & a_{42} \\ a_{51} & a_{52} \end{bmatrix}.$$

While on the subject of submatrices, recall from §1.1.8 that if i and j are scalars, then $A(i, :)$ designates the ith row of A and $A(:, j)$ designates the jth column of A.

1.3.4 Block Matrix Times Vector

An important situation covered by Theorem 1.3.3 is the case of a block matrix times vector. Let us consider the details of the gaxpy $y = Ax + y$ where $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$, and

$$A = \begin{bmatrix} A_1 \\ \vdots \\ A_q \end{bmatrix} \begin{matrix} m_1 \\ \\ m_q \end{matrix} \qquad y = \begin{bmatrix} y_1 \\ \vdots \\ y_q \end{bmatrix} \begin{matrix} m_1 \\ \\ m_q \end{matrix}.$$

We refer to A_i as the ith block row. If $m.vec = (m_1, \dots, m_q)$ is the vector of block row "heights", then from

$$\begin{bmatrix} y_1 \\ \vdots \\ y_q \end{bmatrix} = \begin{bmatrix} A_1 \\ \vdots \\ A_q \end{bmatrix} x + \begin{bmatrix} y_1 \\ \vdots \\ y_q \end{bmatrix}$$

we obtain

$$\begin{array}{l} last = 0 \\ \textbf{for } i = 1{:}q \\ \quad first = last + 1 \\ \quad last = first + m.vec(i) - 1 \\ \quad y(first{:}last) = A(first{:}last, :)x + y(first{:}last) \\ \textbf{end} \end{array} \tag{1.3.1}$$

Each time through the loop an "ordinary" gaxpy is performed so Algorithms 1.1.3 and 1.1.4 apply.

Another way to block the gaxpy computation is to partition A and x as follows:

$$A = \begin{bmatrix} A_1, & \dots, & A_r \end{bmatrix} \qquad x = \begin{bmatrix} x_1 \\ \vdots \\ x_r \end{bmatrix} \begin{matrix} n_1 \\ \\ n_r \end{matrix}.$$
$$\qquad \begin{matrix} n_1 & & n_r \end{matrix}$$

In this case we refer to A_j as the jth block column of A. If $n.vec = (n_1, \dots, n_r)$ is the vector of block column widths, then from

$$y = [A_1, \dots, A_r] \begin{bmatrix} x_1 \\ \vdots \\ x_r \end{bmatrix} + y = \sum_{j=1}^{r} A_j x_j + y$$

we obtain

```
last = 0
for j = 1:r
      first = last + 1
      last = first + n.vec(j) - 1                    (1.3.2)
      y = A(:, first:last)x(first:last) + y
end
```

Again, the gaxpy's performed each time through the loop can be carried out with Algorithm 1.1.3 or 1.1.4.

1.3.5 Block Matrix Multiplication

Just as ordinary, scalar-level matrix multiplication can be arranged in several possible ways, so can the multiplication of block matrices. Different blockings for A, B, and C can set the stage for block versions of the dot product, saxpy, and outer product algorithms of §1.1. To illustrate this with a minimum of subscript clutter, we assume that these three matrices are all n-by-n and that $n = N\ell$ where N and ℓ are positive integers.

If $A = (A_{\alpha\beta})$, $B = (B_{\alpha\beta})$, and $C = (C_{\alpha\beta})$ are N-by-N block matrices with ℓ-by-ℓ blocks, then from Theorem 1.3.3

$$C_{\alpha\beta} = \sum_{\gamma=1}^{N} A_{\alpha\gamma}B_{\gamma\beta} + C_{\alpha\beta} \qquad \alpha = 1{:}N, \quad \beta = 1{:}N.$$

If we organize a matrix multiplication procedure around this summation, then we obtain a block analog of Algorithm 1.1.5:

```
for α = 1:N
      i = (α - 1)ℓ + 1:αℓ
      for β = 1:N
            j = (β - 1)ℓ + 1:βℓ                        (1.3.3)
            for γ = 1:N
                  k = (γ - 1)ℓ + 1:γℓ
                  C(i, j) = A(i, k)B(k, j) + C(i, j)
            end
      end
end
```

Note that if $\ell = 1$, then $\alpha \equiv i$, $\beta \equiv j$, and $\gamma \equiv k$ and we revert to Algorithm 1.1.5.

To obtain a block saxpy matrix multiply, we write $C = AB + C$ as

$$[\, C_1 ,\ldots, C_N \,] = [\, A_1 ,\ldots, A_N \,] \begin{bmatrix} B_{11} & \cdots & B_{1N} \\ \vdots & \ddots & \vdots \\ B_{N1} & \cdots & B_{NN} \end{bmatrix} + [\, C_1 ,\ldots, C_N \,]$$

where $A_\alpha, C_\alpha \in \mathbb{R}^{n\times\ell}$, and $B_{\alpha\beta} \in \mathbb{R}^{\ell\times\ell}$. From this we obtain


```
for β = 1:N
      j = (β - 1)ℓ + 1:βℓ
      for α = 1:N
            i = (α - 1)ℓ + 1:αℓ                        (1.3.4)
            C(:, j) = A(:, i)B(i, j) + C(:, j)
      end
end
```

This is the block version of Algorithm 1.1.7.

A *block outer product* scheme results if we work with the blockings

$$A = [\, A_1 ,\ldots, A_N \,] \qquad B = \begin{bmatrix} B_1^T \\ \vdots \\ B_N^T \end{bmatrix}$$

where $A_\gamma, B_\gamma \in \mathbb{R}^{n\times\ell}$. From Lemma 1.3.2 we have

$$C = \sum_{\gamma=1}^{N} A_\gamma B_\gamma^T + C$$

and so

```
for γ = 1:N
      k = (γ - 1)ℓ + 1:γℓ
      C = A(:, k)B(k, :) + C                        (1.3.5)
end
```

This is the block version of Algorithm 1.1.8.

1.3.6 Complex Matrix Multiplication

Consider the complex matrix multiplication update

$$C_1 + iC_2 = (A_1 + iA_2)(B_1 + iB_2) + (C_1 + iC_2)$$

where all the matrices are real and $i^2 = -1$. Comparing the real and imaginary parts we find

$$\begin{aligned} C_1 &= A_1B_1 - A_2B_2 + C_1 \\ C_2 &= A_1B_2 + A_2B_1 + C_2 \end{aligned}$$

and this can be expressed as follows:

$$\begin{bmatrix} C_1 \\ C_2 \end{bmatrix} = \begin{bmatrix} A_1 & -A_2 \\ A_2 & A_1 \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} + \begin{bmatrix} C_1 \\ C_2 \end{bmatrix}.$$

This suggests how real matrix software might be applied to solve complex matrix problems. The only snag is that the explicit formation of

$$\tilde{A} = \begin{bmatrix} A_1 & -A_2 \\ A_2 & A_1 \end{bmatrix}$$

requires the "double storage" of the matrices A_1 and A_2.

1.3.7 A Divide and Conquer Matrix Multiplication

We conclude this section with a completely different approach to the matrix-matrix multiplication problem. The starting point in the discussion is the 2-by-2 block matrix multiplication

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

where each block is square. In the ordinary algorithm, $C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j}$. There are 8 multiplies and 4 adds. Strassen (1969) has shown how to compute C with just 7 multiplies and 18 adds:

$$\begin{aligned}
P_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
P_2 &= (A_{21} + A_{22})B_{11} \\
P_3 &= A_{11}(B_{12} - B_{22}) \\
P_4 &= A_{22}(B_{21} - B_{11}) \\
P_5 &= (A_{11} + A_{12})B_{22} \\
P_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
P_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \\
C_{11} &= P_1 + P_4 - P_5 + P_7 \\
C_{12} &= P_3 + P_5 \\
C_{21} &= P_2 + P_4 \\
C_{22} &= P_1 + P_3 - P_2 + P_6
\end{aligned}$$

These equations are easily confirmed by substitution. Suppose $n = 2m$ so that the blocks are m-by-m. Counting adds and multiplies in the computation $C = AB$ we find that conventional matrix multiplication involves $(2m)^3$ multiplies and $(2m)^3 - (2m)^2$ adds. In contrast, if Strassen's algorithm is applied *with conventional multiplication at the block level*, then $7m^3$ multiplies and $7m^3 + 11m^2$ adds are required. If $m \gg 1$, then the Strassen method involves about 7/8ths the arithmetic of the fully conventional algorithm.

Now recognize that we can recur on the Strassen idea. In particular, we can apply the Strassen algorithm to each of the half-sized block multiplications associated with the P_i. Thus, if the original A and B are n-by-n and $n = 2^q$, then we can repeatedly apply the Strassen multiplication algorithm. At the bottom "level," the blocks are 1-by-1. Of course, there is no need to

recur down to the $n = 1$ level. When the block size gets sufficiently small, ($n \le n_{min}$), it may be sensible to use conventional matrix multiplication when finding the P_i. Here is the overall procedure:

Algorithm 1.3.1 (Strassen Multiplication) Suppose $n = 2^q$ and that $A \in \mathbb{R}^{n\times n}$ and $B \in \mathbb{R}^{n\times n}$. If $n_{min} = 2^d$ with $d \le q$, then this algorithm computes $C = AB$ by applying Strassen procedure recursively $q - d$ times.

```
function: C = strass(A, B, n, n_min)
    if n ≤ n_min
        C = AB
    else
        m = n/2; u = 1:m; v = m + 1:n;
        P1 = strass(A(u,u) + A(v,v), B(u,u) + B(v,v), m, n_min)
        P2 = strass(A(v,u) + A(v,v), B(u,u), m, n_min)
        P3 = strass(A(u,u), B(u,v) - B(v,v), m, n_min)
        P4 = strass(A(v,v), B(v,u) - B(u,u), m, n_min)
        P5 = strass(A(u,u) + A(u,v), B(v,v), m, n_min)
        P6 = strass(A(v,u) - A(u,u), B(u,u) + B(u,v), m, n_min)
        P7 = strass(A(u,v) - A(v,v), B(v,u) + B(v,v), m, n_min)
        C(u,u) = P1 + P4 - P5 + P7
        C(u,v) = P3 + P5
        C(v,u) = P2 + P4
        C(v,v) = P1 + P3 - P2 + P6
    end
```

Unlike any of our previous algorithms strass is recursive, meaning that it calls itself. Divide and conquer algorithms are often best described in this manner. We have presented this algorithm in the style of a MATLAB function so that the recursive calls can be stated with precision.

The amount of arithmetic associated with **strass** is a complicated function of n and n_{min}. If $n_{min} \gg 1$, then it suffices to count multiplications as the number of additions is roughly the same. If we just count the multiplications, then it suffices to examine the deepest level of the recursion as that is where all the multiplications occur. In **strass** there are $q - d$ subdivisions and thus, 7^{q-d} conventional matrix-matrix multiplications to perform. These multiplications have size n_{min} and thus **strass** involves about $s = (2^d)^3 7^{q-d}$ multiplications compared to $c = (2^q)^3$, the number of multiplications in the conventional approach. Notice that

$$\frac{s}{c} = \left(\frac{2^d}{2^q}\right)^3 7^{q-d} = \left(\frac{7}{8}\right)^{q-d}.$$

If $d = 0$, i.e., we recur on down to the 1-by-1 level, then

$$s = \left(\frac{7}{8}\right)^q c = 7^q = n^{\log_2 7} \approx n^{2.807}.$$

Thus, asymptotically, the number of multiplications in the Strassen procedure is $O(n^{2.807})$. However, the number of additions (relative to the number of multiplications) becomes significant as n_{min} gets small.

Example 1.3.1 If $n = 1024$ and $n_{min} = 64$, then **strass** involves $(7/8)^{10-6} \approx .6$ the arithmetic of the conventional algorithm.

Problems

P1.3.1 Generalize (1.3.3) so that it can handle the variable block-size problem covered by Theorem 1.3.3.

P1.3.2 Generalize (1.3.4) and (1.3.5) so that they can handle the variable block-size case.

P1.3.3 Adapt **strass** so that it can handle square matrix multiplication of any order. Hint: If the "current" A has odd dimension, append a zero row and column.

P1.3.4 Prove that if

$$A = \begin{bmatrix} A_{11} & \cdots & A_{1r} \\ \vdots & \ddots & \vdots \\ A_{q1} & \cdots & A_{qr} \end{bmatrix}$$

is a blocking of the matrix A, then

$$A^T = \begin{bmatrix} A_{11}^T & \cdots & A_{q1}^T \\ \vdots & \ddots & \vdots \\ A_{1r}^T & \cdots & A_{qr}^T \end{bmatrix}.$$

P1.3.5 Suppose n is even and define the following function from $\mathbb{R}^n$ to $\mathbb{R}$:

$$f(x) = x(1{:}2{:}n)^T x(2{:}n) = \sum_{i=1}^{n/2} x_{2i-1}x_{2i}$$

(a) Show that if $x, y \in \mathbb{R}^n$ then

$$x^T y = \sum_{i=1}^{n/2}(x_{2i-1} + y_{2i})(x_{2i} + y_{2i-1}) - f(x) - f(y)$$

(b) Now consider the n-by-n matrix multiplication $C = AB$. Give an algorithm for computing this product that requires $n^3/2$ multiplies once f is applied to the rows of A and the columns of B. See Winograd (1968) for details.

P1.3.6 Prove Lemma 1.3.2 for general s. Hint. Set

$$\rho_\gamma = p_1 + \cdots + p_{\gamma-1} \qquad \gamma = 1{:}s+1$$

and show that

$$c_{ij} = \sum_{\gamma=1}^{s} \sum_{k=\rho_\gamma+1}^{\rho_{\gamma+1}} a_{ik}b_{kj}.$$

P1.3.7 Use Lemmas 1.3.1 and 1.3.2 to prove Theorem 1.3.3. In particular, set

$$A_\gamma = \begin{bmatrix} A_{1\gamma} \\ \vdots \\ A_{q\gamma} \end{bmatrix} \qquad \text{and} \qquad B_\gamma = \begin{bmatrix} B_{\gamma 1} & \cdots & B_{\gamma r} \end{bmatrix}$$

and note from Lemma 1.3.2 that

$$C = \sum_{\gamma=1}^{s} A_\gamma B_\gamma.$$

Now analyze each $A_\gamma B_\gamma$ with the help of Lemma 1.3.1.

Notes and References for Sec. 1.3

For quite some time fast methods for matrix multiplication have attracted a lot of attention within computer science. See

S. Winograd (1968). "A New Algorithm for Inner Product," *IEEE Trans. Comp. C-17*, 693–694.

V. Strassen (1969). "Gaussian Elimination is Not Optimal," *Numer. Math. 13*, 354–356.

V. Pan (1984). "How Can We Speed Up Matrix Multiplication?," *SIAM Review 26*, 393–416.

Many of these methods have dubious practical value. However, with the publication of

D. Bailey (1988). "Extra High Speed Matrix Multiplication on the Cray-2," *SIAM J. Sci. and Stat. Comp. 9*, 603–607.

it is clear that the blanket dismissal of these fast procedures is unwise. The "stability" of the Strassen algorithm is discussed in §2.4.10. See also

N.J. Higham (1990). "Exploiting Fast Matrix Multiplication within the Level 3 BLAS," *ACM Trans. Math. Soft. 16*, 352–368.

C.C. Douglas, M. Heroux, G. Slishman, and R.M. Smith (1994). "GEMMW: A Portable Level 3 BLAS Winograd Variant of Strassen's Matrix-Matrix Multiply Algorithm," *J. Comput. Phys. 110*, 1–10.

1.4 Vectorization and Re-Use Issues

The matrix manipulations discussed in this book are mostly built upon dot products and saxpy operations. *Vector pipeline computers* are able to perform vector operations such as these very fast because of special hardware that is able to exploit the fact that a vector operation is a very regular sequence of scalar operations. Whether or not high performance is extracted from such a computer depends upon the length of the vector operands and a number of other factors that pertain to the movement of data such as vector stride, the number of vector loads and stores, and the level of data re-use. Our goal is to build a useful awareness of these issues. We are *not* trying to build a comprehensive model of vector pipeline

computing that might be used to predict performance. We simply want to identify the kind of *thinking* that goes into the design of an effective vector pipeline code. We do not mention any particular machine. The literature is filled with case studies.

1.4.1 Pipelining Arithmetic Operations

The primary reason why vector computers are fast has to do with *pipelining*. The concept of pipelining is best understood by making an analogy to assembly line production. Suppose the assembly of an individual automobile requires one minute at each of sixty workstations along an assembly line. If the line is well staffed and able to initiate the assembly of a new car every minute, then 1000 cars can be produced from scratch in about 1000 + 60 = 1060 minutes. For a work order of this size the line has an effective "vector speed" of 1000/1060 automobiles per minute. On the other hand, if the assembly line is understaffed and a new assembly can be initiated just once an hour, then 1000 hours are required to produce 1000 cars. In this case the line has an effective "scalar speed" of 1/60th automobile per minute.

So it is with a pipelined vector operation such as the vector add $z = x+y$. The scalar operations $z_i = x_i + y_i$ are the cars. The number of elements is the size of the work order. If the start-to-finish time required for each z_i is τ, then a pipelined, length n vector add could be completed in time much less than $n\tau$. This gives vector speed. Without the pipelining, the vector computation would proceed at a scalar rate and would approximately require time $n\tau$ for completion.

Let us see how a sequence of floating point operations can be pipelined. Floating point operations usually require several cycles to complete. For example, a 3-cycle addition of two scalars x and y may proceed as in FIG. 1.4.1. To visualize the operation, continue with the above metaphor

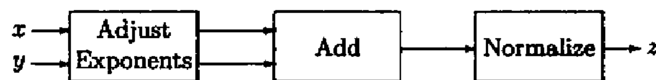


FIG. 1.4.1 *A 3-Cycle Adder*

and think of the addition unit as an assembly line with three "work stations". The input scalars x and y proceed along the assembly line spending one cycle at each of three stations. The sum z emerges after three cycles.

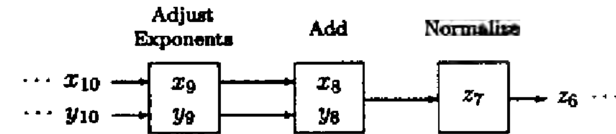


FIG. 1.4.2 *Pipelined Addition*

Note that when a single, "free standing" addition is performed, only one of the three stations is active during the computation.

Now consider a vector addition $z = x + y$. With pipelining, the x and y vectors are streamed through the addition unit. Once the pipeline is filled and steady state reached, a z_i is produced every cycle. In FIG.1.4.2 we depict what the pipeline might look like once this steady state is achieved. In this case, vector speed is about three times scalar speed because the time for an individual add is three cycles.

1.4.2 Vector Operations

A vector pipeline computer comes with a repertoire of *vector instructions*, such as vector add, vector multiply, vector scale, dot product, and saxpy. We assume for clarity that these operations take place in *vector registers*. Vectors travel between the registers and memory by means of *vector load* and *vector store* instructions.

An important attribute of a vector processor is the length of its vector registers which we designate by v_L. A length-n vector operation must be broken down into subvector operations of length v_L or less. Here is how such a partitioning might be managed in the case of a vector addition $z = x + y$ where x and y are n-vectors:

$first = 1$
while $first \leq n$
  $last = \min\{n, first + v_L - 1\}$
  Vector load $x(first{:}last)$.
  Vector load $y(first{:}last)$.
  Vector add: $z(first{:}last) = x(first{:}last) + y(first{:}last)$.
  Vector store $z(first{:}last)$.
  $first = last + 1$
end

A reasonable compiler for a vector computer would automatically generate these vector instructions from a programmer specified $z = x + y$ command.

1.4.3 The Vector Length Issue

Suppose the pipeline for the vector operation *op* takes τ_{op} cycles to "set up." Assume that one component of the result is obtained per cycle once the pipeline is filled. The time required to perform an n-dimensional *op* is then given by

$$T_{op}(n) = (\tau_{op} + n)\mu \qquad n \le v_L$$

where μ is the cycle time and v_L is the length of the vector hardware.

If the vectors to be combined are longer than the vector hardware length, then as we have seen the overall vector operation must be broken down into hardware-manageable chunks. Thus, if

$$n = n_1 v_L + n_0 \qquad 0 \le n_0 < v_L \; ,$$

then we assume that

$$T_{op}(n) = \begin{cases} n_1(\tau_{op} + v_L)\mu & n_0 = 0 \\ (n_1(\tau_{op} + v_L) + \tau_{op} + n_0)\mu & n_0 \neq 0 \end{cases}$$

specifies the overall time required to perform a length-n *op*. This simplifies to

$$T_{op}(n) = (n + \tau_{op}\mathrm{ceil}(n/v_L))\,\mu$$

where ceil(α) is the smallest integer such that $\alpha \le \mathrm{ceil}(\alpha)$. If ρ flops per component are involved, then the effective rate of computation for general n is given by

$$R_{op}(n) = \frac{\rho n}{T_{op}(n)} = \frac{\rho}{\mu}\,\frac{1}{1 + \frac{\tau_{op}}{n}\,\mathrm{ceil}\left(\frac{n}{v_L}\right)} .$$

(If μ is in seconds, then R_{op} is in flops per second.) The asymptotic rate of performance is given by

$$\lim_{n\to\infty} R_{op}(n) = \frac{1}{1 + \frac{\tau_{op}}{v_L}}\,\frac{\rho}{\mu} .$$

As a way of assessing how serious the start-up overhead is for a vector operation, Hockney and Jesshope (1988) define the quantity $n_{1/2}$ to be the smallest n for which half of peak performance is achieved, i.e.,

$$\frac{\rho n_{1/2}}{T_{op}(n_{1/2})} = \frac{1}{2}\frac{\rho}{\mu} .$$

Machines that have big $n_{1/2}$ factors do not perform well on short vector operations.

Let us see what the above performance model says about the design of the matrix multiply update $C = AB + C$ where $A \in \mathbb{R}^{m\times p}$, $B \in \mathbb{R}^{p\times n}$, and $C \in \mathbb{R}^{m\times n}$. Recall from §1.1.11 that there are six possible versions of the conventional algorithm and they correspond to the six possible loop orderings of


```
for i = 1:m
    for j = 1:n
        for k = 1:p
            C(i,j) = A(i,k)B(k,j) + C(i,j)
        end
    end
end
```

This is the ijk variant and its innermost loop oversees a length-p dot product. Thus, our performance model predicts that

$$T_{ijk} = mnp + mn \cdot \mathrm{ceil}(p/v_L)\tau_{dot}$$

cycles are required. A similar analysis for each of the other variants leads to the following table:

Variant	Cycles
ijk	$mnp + mn \cdot \tau_{dot}(p/v_L)$
jik	$mnp + mn \cdot \tau_{dot}(p/v_L)$
ikj	$mnp + mp \cdot \tau_{sax}(n/v_L)$
jki	$mnp + np \cdot \tau_{sax}(m/v_L)$
kij	$mnp + mp \cdot \tau_{sax}(n/v_L)$
kji	$mnp + np \cdot \tau_{sax}(m/v_L)$

We make a few observations based upon some elementary integer arithmetic manipulation. Assume that τ_{sax} and τ_{dot} are roughly equal. If m, n, and p are all less than v_L, then the most efficient variants will have the longest inner loops. If m, n, and p are much bigger than v_L, then the distinction between the six options is small.

1.4.4 The Stride Issue

The "layout" of a vector operand in memory often has a bearing on execution speed. The key factor is stride. The *stride* of a stored floating point vector is the distance (in logical memory locations) between the vector's components. Accessing a row in a two-dimensional Fortran array is not a unit stride operation because arrays are stored by column. In C, it is just the opposite as matrices are stored by row. Nonunit stride vector operations may interfere with the pipelining capability of a computer degrading performance.

To clarify the stride issue we consider how the six variants of matrix multiplication "pull up" data from the A, B, and C matrices in the inner loop. This is where the vector calculation occurs (dot product or saxpy) and there are three possibilities:

jki or *kji*:

```
for i = 1:m
    C(i,j) = C(i,j) + A(i,k)B(k,j)
end
```

ikj or *kij*:

```
for j = 1:n
    C(i,j) = C(i,j) + A(i,k)B(k,j)
end
```

ijk or *jik*:

```
for k = 1:p
    C(i,j) = C(i,j) + A(i,k)B(k,j)
end
```

Here is a table that specifies the A, B, and C strides associated with each of these possibilities:

Variant	A Stride	B Stride	C Stride
jki or *kji*	Unit	0	Unit
ikj or *kij*	0	Non-Unit	Non-Unit
ijk or *jik*	Non-Unit	Unit	0

Storage in column-major order is assumed. A stride of zero means that only a single array element is accessed in the inner loop. From the stride point of view, it is clear that we should favor the *jki* and *kji* variants. This may not coincide with a preference that is based on vector length considerations. Dilemmas of this type are typical in high performance computing. One goal (maximize vector length) can conflict with another (impose unit stride).

Sometimes a vector stride/vector length conflict can s be resolved through the intelligent choice of data structures. Consider the gaxpy $y = Ax + y$ where $A \in \mathbb{R}^{n\times n}$ is symmetric. Assume that $n \le v_L$ for simplicity. If A is stored conventionally and Algorithm 1.1.4 is used, then the central computation entails n, unit stride saxpy's each having length n:

```
for j = 1:n
    y = A(:,j)x(j) + y
end
```

Our simple execution model tells us that

$$T_1 = n(\tau_{sax} + n)$$

cycles are required.

In §1.2.7 we introduced the lower triangular storage scheme for symmetric matrices and obtained this version of the gaxpy:


```
for j = 1:n
    for i = 1:j-1
        y(i) = A.vec((i-1)n - i(i-1)/2 + j)x(j) + y(i)
    end
    for i = j:n
        y(i) = A.vec((j-1)n - j(j-1)/2 + i)x(j) + y(i)
    end
end
```

Notice that the first i-loop does *not* define a unit stride saxpy. If we assume that a length n, nonunit stride saxpy is equivalent to n unit-length saxpys (a worst case scenario), then this implementation involves

$$T_2 = n\left(\frac{n}{2}\tau_{sax} + n\right)$$

cycles.

In §1.2.8 we developed the store-by-diagonal version:

```
for i = 1:n
    y(i) = A.diag(i)x(i) + y(i)
end
for k = 1:n-1
    t = nk - k(k-1)/2
    {y = D(A,k)x + y}
    for i = 1:n-k
        y(i) = A.diag(i+t)x(i+k) + y(i)
    end
    {y = D(A,k)^T x + y}
    for i = 1:n-k
        y(i+k) = A.diag(i+t)x(i) + y(i+k)
    end
end
```

In this case both inner loops define a unit stride vector multiply (vm) and our model of execution predicts

$$T_3 = n\,(2\tau_{vm} + n)$$

cycles.

The example shows how the choice of data structure can effect the stride attributes of an algorithm. Store by diagonal seems attractive because it represents the matrix compactly and has unit stride. However, a careful which-is-best analysis would depend upon the values of τ_{sax} and τ_{vm} and the precise penalties for nonunit stride computation and excess storage. The complexity of the situation would call for careful benchmarking.

1.4.5 Thinking About Data Motion

Another important attribute of a matrix algorithm concerns the actual volume of data that has to be moved around during execution. Matrices sit in memory but the computations that involve their entries take place in functional units. The control of memory traffic is crucial to performance in many computers. To continue with the factory metaphor used at the beginning of this section: *Can we keep the superfast arithmetic units busy with enough deliveries of matrix data and can we ship the results back to memory fast enough to avoid backlog?* FIG.1.4.3 depicts the typical situation in an advanced uniprocessor environment. Details vary from machine

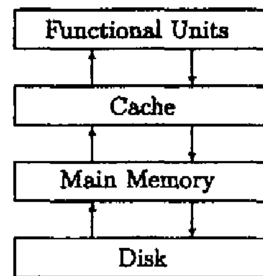


FIG. 1.4.3 *Memory Hierarchy*

to machine, but two "axioms;; prevail:

- Each level in the hierarchy has a limited capacity and for economic reasons this capacity is usually smaller as we ascend the hierarchy.
- There is a cost, sometimes relatively great, associated with the moving of data between two levels in the hierarchy.

The design of an efficient matrix algorithm requires careful thinking about the flow of data in between the various levels of storage. The vector touch and data re-use issues are important in this regard.

1.4.6 The Vector Touch Issue

In many advanced computers, data is moved around in chunks, e.g., vectors. The time required to read or write a vector to memory is comparable to the time required to engage the vector in a dot product or saxpy. Thus, the number of *vector touches* associated with a matrix code is a very important statistic. By a "vector touch" we mean either a vector load or store.

Let's count the number of vector touches associated with an m-by-n outer product. Assume that $m = m_1 v_L$ and $n = n_1 v_L$ where v_L is the vector hardware length. (See §1.4.3.) In this environment, the outer product update $A = A + xy^T$ would be arranged as follows:

```
for α = 1:m_1
    i = (α − 1)v_L + 1:αv_L
    for β = 1:n_1
        j = (β − 1)v_L + 1:βv_L
        A(i,j) = A(i,j) + x(i)y(j)^T
    end
end
```

Each column of the submatrix $A(i,j)$ must be loaded, updated, and then stored. Not forgetting to account for the vector touches associated with x and y we see that approximately

$$\sum_{\alpha=1}^{m_1}\left(1+\sum_{\beta=1}^{n_1}(1+2v_L)\right) \approx 2m_1 n$$

vector touches are required. (Low order terms do not contribute to the analysis.)

Now consider the gaxpy update $y = Ax + y$ where $y \in \mathbb{R}^m$, $x \in \mathbb{R}^n$ and $A \in \mathbb{R}^{m\times n}$. Breaking this computation down into segments of length v_L gives

```
for α = 1:m_1
    i = (α − 1)v_L + 1:αv_L
    for β = 1:n_1
        j = (β − 1)v_L + 1:βv_L
        y(i) = y(i) + A(i,j)x(j)
    end
end
```

Again, each column of submatrix $A(i,j)$ must be read but the only writing to memory involves subvectors of y. Thus, the number of vector touches for an m-by-n gaxpy is

$$\sum_{\alpha=1}^{m_1}\left(2+\sum_{\beta=1}^{n_1}(1+v_L)\right) \approx m_1 n .$$

This is half the number required by an identically-sized the outer product. Thus, if a computation can be arranged in terms of either outer products or gaxpys, then the former is preferable from the vector touch standpoint.

1.4.7 Blocking and Re-Use

A *cache* is a small high-speed memory situated in between the functional units and main memory. See FIG.1.4.3. Cache utilization colors performance because it has a direct bearing upon how data flows in between the functional units and the lower levels of memory.

To illustrate this we consider the computation of the matrix multiply update $C = AB + C$ where $A, B, C \in \mathbb{R}^{n\times n}$ reside in main memory[2]. All data must pass through the cache on its way to the functional units where the floating point computations are carried out. If the cache is small and n is big, then the update must be broken down into smaller parts so that the cache can "gracefully" process the flow of data.

One strategy is to block the B and C matrices,

$$B = [\ \underset{\ell}{B_1}, \ldots, \underset{\ell}{B_N}\ ] \qquad C = [\ \underset{\ell}{C_1}, \ldots, \underset{\ell}{C_N}\ ]$$

where we assume that $n = \ell N$. From the expansion

$$C_\alpha = AB_\alpha + C_\alpha = \sum_{k=1}^{n} A(:,k)B_\alpha(k,:) + C_\alpha$$

we obtain the following computational framework:

for $\alpha = 1{:}N$
  Load B_α and C_α into cache.
  for $k = 1{:}n$
    Load $A(:,k)$ into cache and update C_α:
      $C_\alpha = A(:,k)B_\alpha(k,:) + C_\alpha$
  end
  Store C_α in main memory.
end

Note that if M is the cache size measured in floating point words, then we must have

$$2n\ell + n \le M. \tag{1.4.1}$$

Let Γ_1 be the number of floating point numbers that flow (in either direction) between cache and main memory. Note that every entry in B is loaded into cache once, every entry in C is loaded into cache once and stored back in main memory once, and every entry in A is loaded into cache $N = n/\ell$ times. It follows that

$$\Gamma_1 = 3n^2 + \frac{n^3}{\ell}.$$

[2]The discussion which follows would also apply if the matrices were on a disk and needed to be brought into main memory.

In the interest of keeping data motion to a minimum, we choose ℓ to be as large as possible subject to the constraint (1.4.1). We therefore set

$$\ell \approx \frac{1}{2}\left(\frac{M}{n} - 1\right)$$

obtaining

$$\Gamma_1 \approx 3n^2 + \frac{2n^4}{M-n}.$$

(We use "$\approx$" to emphasize the approximate nature of our analysis.) If cache is large enough to house the entire B and C matrices with room left over for a column of A, then $\ell = n$ and $\Gamma_1 = 4n^2$. At the other extreme, if we can just fit three columns in cache, then $\ell = 1$ and $\Gamma_1 \approx n^3$.

Now let us regard $A = (A_{\alpha\beta})$, $B = (B_{\alpha\beta})$, and $C = (C_{\alpha\beta})$ as N-by-N block matrices with uniform block size $\ell = n/N$. With this blocking the computation of

$$C_{\alpha\beta} = \sum_{\gamma=1}^{N} A_{\alpha\gamma}B_{\gamma\beta} \qquad \alpha = 1{:}N,\ \beta = 1{:}N$$

can be arranged as follows:

for $\alpha = 1{:}N$
  for $\beta = 1{:}N$
    Load $C_{\alpha\beta}$ into cache.
    for $\gamma = 1{:}N$
      Load $A_{\alpha\gamma}$ and $B_{\gamma\beta}$ into cache.
      $C_{\alpha\beta} = C_{\alpha\beta} + A_{\alpha\gamma}B_{\gamma\beta}$
    end
    Store $C_{\alpha\beta}$ in main memory.
  end
end

In this case the main memory/cache traffic sums to

$$\Gamma_2 = 2n^2 + \frac{2n^3}{\ell}$$

because each entry in A and B is loaded $N = n/\ell$ times and each entry in C is loaded once and stored once. We can minimize this by choosing ℓ to be as large as possible subject to the constraint that three blocks fit in cache, i.e.,

$$3\ell^2 \le M$$

Setting $\ell \approx \sqrt{M/3}$ gives

$$\Gamma_2 \approx 2n^2 + 2n^3\sqrt{\frac{3}{M}}.$$

A manipulation shows that

$$\frac{\Gamma_1}{\Gamma_2} \approx \frac{3n^2 + \frac{2n^4}{M-n}}{2n^2 + 2n^3\sqrt{\frac{3}{M}}} \geq \frac{3 + 2\frac{n^2}{M}}{2 + 2\sqrt{3}\sqrt{\frac{n^2}{M}}}.$$

The key quantity here is n^2/M, the ratio of matrix size (in floating point words) to cache size. As this ratio grows the we find that

$$\frac{\Gamma_1}{\Gamma_2} \approx \frac{n}{\sqrt{3M}}$$

showing that the second blocking strategy is superior from the standpoint of data motion to and from the cache. The fundamental conclusion to be reached from all of this is that *blocking effects data motion.*

1.4.8 Block Matrix Data Structures

We conclude this section with a discussion about block data structures. A programming language that supports two-dimensional arrays must have a convention for storing such a structure in memory. For example, Fortran stores two-dimensional arrays in *column major order.* This means that the entries within a column are contiguous in memory. Thus, if 24 storage locations are allocated for $A \in \mathbb{R}^{4\times 6}$, then in traditional store-by-column format the matrix entries are "lined up" in memory as depicted in FIG. 1.4.4. In other words, if $A \in \mathbb{R}^{m\times n}$ is stored in $v(1{:}mn)$, then we identify

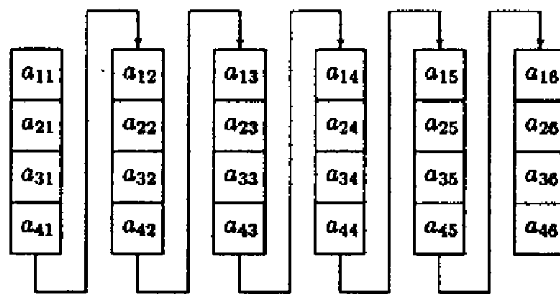


FIG. 1.4.4 *Store by Column (4-by-6 case)*

$A(i,j)$ with $v((j-1)m+i)$. For algorithms that access matrix data by column this is a good arrangement since the column entries are contiguous in memory.

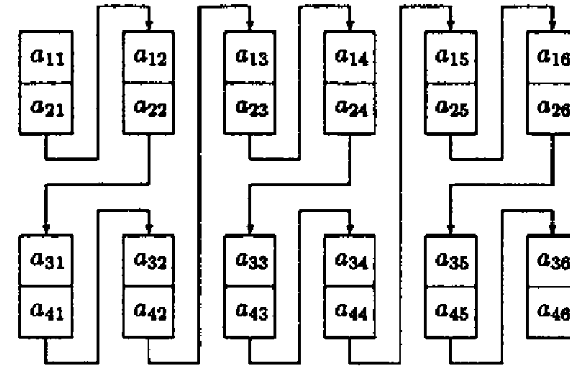


FIG. 1.4.5 *Store-by-Blocks (4-by-6 case with 2-by-2 Blocks)*

In certain block matrix algorithms it is sometimes useful to store matrices by blocks rather than by column. Suppose, for example, that the matrix A above is a 2-by-3 block matrix with 2-by-2 blocks. In a store-by-column block scheme with store-by-column within each block, the 24 entries are arranged in memory as shown in FIG. 1.4.5. This data structure can be attractive for block algorithms because the entries within a given block are contiguous in memory.

Problems

P1.4.1 Consider the matrix product $D = ABC$ where $A \in \mathbb{R}^{m\times r}$, $B \in \mathbb{R}^{r\times n}$ and $C \in \mathbb{R}^{n\times q}$. Assume that all the matrices are stored by column and that the time required to execute a unit-stride saxpy operation of length k is of the form $t(k) = (L+k)\mu$ where L is a constant and μ is the cycle time. Based on this model, when is it more economical to compute D as $D = (AB)C$ instead of as $D = A(BC)$? Assume that all matrix multiplies are done using the jki, (gaxpy) algorithm.

P1.4.2 What is the total time spent in jki variant on the saxpy operations assuming that all the matrices are stored by column and that the time required to execute a unit-stride saxpy operation of length k is of the form $t(k) = (L+k)\mu$ where L is a constant and μ is the cycle time? Specialize the algorithm so that it efficiently handles the case when A and B are n-by-n and upper triangular. Does it follow that the triangular implementation is six times faster as the flop count suggests?

P1.4.3 Give an algorithm for computing $C = A^TBA$ where A and B are n–by–n and B is symmetric. Arrays should be accessed in unit stride fashion within all innermost loops.

P1.4.4 Suppose $A \in \mathbb{R}^{m\times n}$ is stored by column in $A.col(1{:}mn)$. Assume that $m = \ell_1 M$

and $n = \ell_2 N$ and that we regard A as an M-by-N block matrix with ℓ_1-by-ℓ_2 blocks. Given i, j, α, and β that satisfy $1 \le i \le \ell_1$, $1 \le j \le \ell_2$, $1 \le \alpha \le M$, and $1 \le \beta \le N$ determine k so that $A.col(k)$ houses the (i,j) entry of $A_{\alpha\beta}$. Give an algorithm that overwrites $A.col$ with A stored by block as in Figure 1.4.5. How big of a work array is required?

Notes and References for Sec. 1.4

Two excellent expositions about vector computation are

J.J. Dongarra, F.G. Gustavson, and A. Karp (1984). "Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine," *SIAM Review 26*, 91-112.

J.M. Ortega and R.G. Voigt (1985). "Solution of Partial Differential Equations on Vector and Parallel Computers," *SIAM Review 27*, 149-240.

A very detailed look at matrix computations in hierarchical memory systems can be found in

K. Gallivan, W. Jalby, U. Meier, and A.H. Sameh (1988). "Impact of Hierarchical Memory Systems on Linear Algebra Algorithm Design," *Int'l J. Supercomputer Applic. 2*, 12-48.

See also

W. Schönauer (1987). *Scientific Computing on Vector Computers*, North Holland, Amsterdam.

R.W. Hockney and C.R. Jesshope (1988). *Parallel Computers 2*, Adam Hilger, Bristol and Philadelphia.

where various models of vector processor performance are set forth. Papers on the practical aspects of vector computing include

J.J. Dongarra and A. Hinds (1979). "Unrolling Loops in Fortran," *Software Practice and Experience 9*, 219-229.

J.J. Dongarra and S. Eisenstat (1984). "Squeezing the Most Out of an Algorithm in Cray Fortran," *ACM Trans. Math. Soft. 10*, 221-230.

B.L. Buzbee (1986) "A Strategy for Vectorization," *Parallel Computing 3*, 187-192.

K. Gallivan, W. Jalby, and U. Meier (1987). "The Use of BLAS3 in Linear Algebra on a Parallel Processor with a Hierarchical Memory," *SIAM J. Sci. and Stat. Comp. 8*, 1079-1084.

J.J. Dongarra and D. Walker (1995). "Software Libraries for Linear Algebra Computations on High Performance Computers," *SIAM Review 37*, 151-180.

Chapter 2

Matrix Analysis

§2.1 Basic Ideas from Linear Algebra
§2.2 Vector Norms
§2.3 Matrix Norms
§2.4 Finite Precision Matrix Computations
§2.5 Orthogonality and the SVD
§2.6 Projections and the CS Decomposition
§2.7 The Sensitivity of Square Linear Systems

The analysis and derivation of algorithms in the matrix computation area requires a facility with certain aspects of linear algebra. Some of the basics are reviewed in §2.1. Norms and their manipulation are covered in §2.2 and §2.3. In §2.4 we develop a model of finite precision arithmetic and then use it in a typical roundoff analysis.

The next two sections deal with orthogonality, which has a prominent role to play in matrix computations. The singular value decomposition and the CS decomposition are a pair of orthogonal reductions that provide critical insight into the important notions of rank and distance between subspaces. In §2.7 we examine how the solution of a linear system $Ax = b$ changes if A and b are perturbed. The important concept of matrix condition is introduced.

Before You Begin

References that complement this chapter include Forsythe and Moler (1967), Stewart (1973), Stewart and Sun (1990), and Higham (1996).

2.1 Basic Ideas from Linear Algebra

This section is a quick review of linear algebra. Readers who wish a more detailed coverage should consult the references at the end of the section.

2.1.1 Independence, Subspace, Basis, and Dimension

A set of vectors $\{a_1,\ldots,a_n\}$ in $\mathbb{R}^m$ is *linearly independent* if $\sum_{j=1}^n \alpha_j a_j = 0$ implies $\alpha(1{:}n) = 0$. Otherwise, a nontrivial combination of the a_i is zero and $\{a_1,\ldots,a_n\}$ is said to be *linearly dependent*.

A *subspace* of $\mathbb{R}^m$ is a subset that is also a vector space. Given a collection of vectors $a_1,\ldots,a_n \in \mathbb{R}^m$, the set of all linear combinations of these vectors is a subspace referred to as the *span* of $\{a_1,\ldots,a_n\}$:

$$\text{span}\{a_1,\ldots,a_n\} = \left\{ \sum_{j=1}^n \beta_j a_j : \beta_j \in \mathbb{R} \right\}.$$

If $\{a_1,\ldots,a_n\}$ is independent and $b \in \text{span}\{a_1,\ldots,a_n\}$, then b is a unique linear combination of the a_j.

If $S_1,\ldots,S_k$ are subspaces of $\mathbb{R}^m$, then their sum is the subspace defined by $S = \{ a_1 + a_2 + \cdots + a_k : a_i \in S_i,\ i = 1{:}k \}$. S is said to be a *direct sum* if each $v \in S$ has a unique representation $v = a_1 + \cdots + a_k$ with $a_i \in S_i$. In this case we write $S = S_1 \oplus \cdots \oplus S_k$. The intersection of the S_i is also a subspace, $S = S_1 \cap S_2 \cap \cdots \cap S_k$.

The subset $\{a_{i_1},\ldots,a_{i_k}\}$ is a *maximal linearly independent subset* of $\{a_1,\ldots,a_n\}$ if it is linearly independent and is not properly contained in any linearly independent subset of $\{a_1,\ldots,a_n\}$. If $\{a_{i_1},\ldots,a_{i_k}\}$ is maximal, then $\text{span}\{a_1,\ldots,a_n\} = \text{span}\{a_{i_1},\ldots,a_{i_k}\}$ and $\{a_{i_1},\ldots,a_{i_k}\}$ is a *basis* for $\text{span}\{a_1,\ldots,a_n\}$. If $S \subseteq \mathbb{R}^m$ is a subspace, then it is possible to find independent basic vectors $a_1,\ldots,a_k \in S$ such that $S = \text{span}\{a_1,\ldots,a_k\}$. All bases for a subspace S have the same number of elements. This number is the *dimension* and is denoted by $\dim(S)$.

2.1.2 Range, Null Space, and Rank

There are two important subspaces associated with an m-by-n matrix A. The *range* of A is defined by

$$\text{ran}(A) = \{y \in \mathbb{R}^m : y = Ax \text{ for some } x \in \mathbb{R}^n\},$$

and the *null space* of A is defined by

$$\text{null}(A) = \{x \in \mathbb{R}^n : Ax = 0\}.$$

If $A = [\, a_1,\ldots,a_n \,]$ is a column partitioning, then

$$\text{ran}(A) = \text{span}\{a_1,\ldots,a_n\}.$$

The *rank* of a matrix A is defined by

$$\text{rank}(A) = \dim(\text{ran}(A)).$$

It can be shown that $\text{rank}(A) = \text{rank}(A^T)$. We say that $A \in \mathbb{R}^{m\times n}$ is *rank deficient* if $\text{rank}(A) < \min\{m,n\}$. If $A \in \mathbb{R}^{m\times n}$, then

$$\dim(\text{null}(A)) + \text{rank}(A) = n.$$

2.1.3 Matrix Inverse

The n-by-n *identity matrix* I_n is defined by the column partitioning

$$I_n = [\, e_1,\ldots,e_n \,]$$

where e_k is the kth "canonical" vector:

$$e_k = (\underbrace{0,\ldots,0}_{k-1}, 1, \underbrace{0,\ldots,0}_{n-k})^T.$$

The canonical vectors arise frequently in matrix analysis and if their dimension is ever ambiguous, we use superscripts, i.e., $e_k^{(n)} \in \mathbb{R}^n$.

If A and X are in $\mathbb{R}^{n\times n}$ and satisfy $AX = I$, then X is the *inverse* of A and is denoted by A^{-1}. If A^{-1} exists, then A is said to be *nonsingular*. Otherwise, we say A is *singular*.

Several matrix inverse properties have an important role to play in matrix computations. The inverse of a product is the reverse product of the inverses:

$$(AB)^{-1} = B^{-1}A^{-1}. \tag{2.1.1}$$

The transpose of the inverse is the inverse of the transpose:

$$(A^{-1})^T = (A^T)^{-1} \equiv A^{-T}. \tag{2.1.2}$$

The identity

$$B^{-1} = A^{-1} - B^{-1}(B - A)A^{-1} \tag{2.1.3}$$

shows how the inverse changes if the matrix changes.

The *Sherman-Morrison-Woodbury formula* gives a convenient expression for the inverse of $(A + UV^T)$ where $A \in \mathbb{R}^{n\times n}$ and U and V are n-by-k:

$$(A + UV^T)^{-1} = A^{-1} - A^{-1}U(I + V^TA^{-1}U)^{-1}V^TA^{-1}. \tag{2.1.4}$$

A rank k correction to a matrix results in a rank k correction of the inverse. In (2.1.4) we assume that both A and $(I + V^TA^{-1}U)$ are nonsingular.

Any of these facts can be verified by just showing that the "proposed" inverse does the job. For example, here is how to confirm (2.1.3):

$$B\left(A^{-1} - B^{-1}(B - A)A^{-1}\right) = BA^{-1} - (B - A)A^{-1} = I.$$

2.1.4 The Determinant

If $A = (\alpha) \in \mathbb{R}^{1\times 1}$, then its *determinant* is given by $\det(A) = \alpha$. The determinant of $A \in \mathbb{R}^{n\times n}$ is defined in terms of order $n-1$ determinants:

$$\det(A) = \sum_{j=1}^n (-1)^{j+1} a_{1j} \det(A_{1j}).$$

Here, A_{1j} is an $(n-1)$-by-$(n-1)$ matrix obtained by deleting the first row and jth column of A. Useful properties of the determinant include

$$\begin{array}{rcll} \det(AB) & = & \det(A)\det(B) & A, B \in \mathbb{R}^{n\times n} \\ \det(A^T) & = & \det(A) & A \in \mathbb{R}^{n\times n} \\ \det(cA) & = & c^n\det(A) & c \in \mathbb{R}, A \in \mathbb{R}^{n\times n} \\ \det(A) \neq 0 & \Leftrightarrow & A \text{ is nonsingular} & A \in \mathbb{R}^{n\times n} \end{array}$$

2.1.5 Differentiation

Suppose α is a scalar and that $A(\alpha)$ is an m-by-n matrix with entries $a_{ij}(\alpha)$. If $a_{ij}(\alpha)$ is a differentiable function of α for all i and j, then by $\dot{A}(\alpha)$ we mean the matrix

$$\dot{A}(\alpha) = \frac{d}{d\alpha}A(\alpha) = \left(\frac{d}{d\alpha}a_{ij}(\alpha)\right) = (\dot{a}_{ij}(\alpha)).$$

The differentiation of a parameterized matrix turns out to be a handy way to examine the sensitivity of various matrix problems.

Problems

P2.1.1 Show that if $A \in \mathbb{R}^{m\times n}$ has rank p, then there exists an $X \in \mathbb{R}^{m\times p}$ and a $Y \in \mathbb{R}^{n\times p}$ such that $A = XY^T$, where $\text{rank}(X) = \text{rank}(Y) = p$.

P2.1.2 Suppose $A(\alpha) \in \mathbb{R}^{m\times r}$ and $B(\alpha) \in \mathbb{R}^{r\times n}$ are matrices whose entries are differentiable functions of the scalar α. Show

$$\frac{d}{d\alpha}[A(\alpha)B(\alpha)] = \left[\frac{d}{d\alpha}A(\alpha)\right]B(\alpha) + A(\alpha)\left[\frac{d}{d\alpha}B(\alpha)\right].$$

P2.1.3 Suppose $A(\alpha) \in \mathbb{R}^{n\times n}$ has entries that are differentiable functions of the scalar α. Assuming $A(\alpha)$ is always nonsingular, show

$$\frac{d}{d\alpha}\left[A(\alpha)^{-1}\right] = -A(\alpha)^{-1}\left[\frac{d}{d\alpha}A(\alpha)\right]A(\alpha)^{-1}.$$

P2.1.4 Suppose $A \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$ and that $\phi(x) = \frac{1}{2}x^TAx - x^Tb$. Show that the gradient of ϕ is given by $\nabla\phi(x) = \frac{1}{2}(A^T + A)x - b$.

P2.1.5 Assume that both A and $A+uv^T$ are nonsingular where $A \in \mathbb{R}^{n\times n}$ and $u, v \in \mathbb{R}^n$. Show that if x solves $(A + uv^T)x = b$, then it also solves a perturbed right hand side problem of the form $Ax = b + \alpha u$. Give an expression for α in terms of A, u, and v.

Notes and References for Sec. 2.1

There are many introductory linear algebra texts. Among them, the following are particularly useful:

P.R. Halmos (1958). *Finite Dimensional Vector Spaces*, 2nd ed., Van Nostrand-Reinhold, Princeton.

S.J. Leon (1980). *Linear Algebra with Applications*. Macmillan, New York.
G. Strang (1993). *Introduction to Linear Algebra*, Wellesley-Cambridge Press, Wellesley MA.
D. Lay (1994). *Linear Algebra and Its Applications*, Addison-Wesley, Reading, MA.
C. Meyer (1997). *A Course in Applied Linear Algebra*, SIAM Publications, Philadelphia, PA.

More advanced treatments include Gantmacher (1959), Horn and Johnson (1985, 1991), and

A.S. Householder (1964). *The Theory of Matrices in Numerical Analysis*, Ginn (Blaisdell), Boston.
M. Marcus and H. Minc (1964). *A Survey of Matrix Theory and Matrix Inequalities*, Allyn and Bacon, Boston.
J.N. Franklin (1968). *Matrix Theory* Prentice Hall, Englewood Cliffs, NJ.
R. Bellman (1970). *Introduction to Matrix Analysis, Second Edition*, McGraw-Hill, New York.
P. Lancaster and M. Tismenetsky (1985). *The Theory of Matrices, Second Edition*, Academic Press, New York.
J.M. Ortega (1987). *Matrix Theory: A Second Course*, Plenum Press, New York.

2.2 Vector Norms

Norms serve the same purpose on vector spaces that absolute value does on the real line: they furnish a measure of distance. More precisely, $\mathbb{R}^n$ together with a norm on $\mathbb{R}^n$ defines a metric space. Therefore, we have the familiar notions of neighborhood, open sets, convergence, and continuity when working with vectors and vector-valued functions.

2.2.1 Definitions

A *vector norm* on $\mathbb{R}^n$ is a function $f:\mathbb{R}^n \to \mathbb{R}$ that satisfies the following properties:

$$\begin{array}{lll} f(x) \geq 0 & x \in \mathbb{R}^n, & (f(x) = 0 \text{ iff } x = 0) \\ f(x+y) \leq f(x) + f(y) & x, y \in \mathbb{R}^n & \\ f(\alpha x) = |\alpha| f(x) & \alpha \in \mathbb{R}, x \in \mathbb{R}^n & \end{array}$$

We denote such a function with a double bar notation: $f(x) = \| x \|$. Subscripts on the double bar are used to distinguish between various norms.

A useful class of vector norms are the *p-norms* defined by

$$\| x \|_p = (|x_1|^p + \cdots + |x_n|^p)^{\frac{1}{p}} \qquad p \geq 1. \tag{2.2.1}$$

Of these the 1, 2, and ∞ norms are the most important:

$$\begin{array}{rcl} \| x \|_1 & = & |x_1| + \cdots + |x_n| \\ \| x \|_2 & = & \left(|x_1|^2 + \cdots + |x_n|^2\right)^{\frac{1}{2}} = \left(x^Tx\right)^{\frac{1}{2}} \\ \| x \|_\infty & = & \max\limits_{1\leq i\leq n} |x_i| \end{array}$$

A *unit vector* with respect to the norm $\| \cdot \|$ is a vector x that satisfies $\| x \| = 1$.

2.2.2 Some Vector Norm Properties

A classic result concerning p-norms is the *Holder inequality:*

$$|x^T y| \;\leq\; \| x \|_p \| y \|_q \qquad \frac{1}{p} + \frac{1}{q} = 1. \tag{2.2.2}$$

A very important special case of this is the *Cauchy-Schwartz inequality:*

$$|x^T y| \;\leq\; \| x \|_2 \| y \|_2. \tag{2.2.3}$$

All norms on $\mathbb{R}^n$ are *equivalent* , i.e., if $\| \cdot \|_\alpha$ and $\| \cdot \|_\beta$ are norms on $\mathbb{R}^n$, then there exist positive constants, c_1 and c_2 such that

$$c_1 \| x \|_\alpha \leq \| x \|_\beta \leq c_2 \| x \|_\alpha \tag{2.2.4}$$

for all $x \in \mathbb{R}^n$. For example, if $x \in \mathbb{R}^n$, then

$$\| x \|_2 \;\leq\; \| x \|_1 \;\leq\; \sqrt{n}\, \| x \|_2 \tag{2.2.5}$$

$$\| x \|_\infty \;\leq\; \| x \|_2 \;\leq\; \sqrt{n}\, \| x \|_\infty \tag{2.2.6}$$

$$\| x \|_\infty \;\leq\; \| x \|_1 \;\leq\; n \| x \|_\infty. \tag{2.2.7}$$

2.2.3 Absolute and Relative Error

Suppose $\hat{x} \in \mathbb{R}^n$ is an approximation to $x \in \mathbb{R}^n$. For a given vector norm $\| \cdot \|$ we say that

$$\epsilon_{abs} \;=\; \| \hat{x} - x \|$$

is the *absolute error* in $\hat{x}$. If $x \neq 0$, then

$$\epsilon_{rel} \;=\; \frac{\| \hat{x} - x \|}{\| x \|}$$

prescribes the *relative error* in $\hat{x}$. Relative error in the ∞-norm can be translated into a statement about the number of correct significant digits in $\hat{x}$. In particular, if

$$\frac{\| \hat{x} - x \|_\infty}{\| x \|_\infty} \;\approx\; 10^{-p},$$

then the largest component of $\hat{x}$ has approximately p correct significant digits.

Example 2.2.1 If $x = (1.234 \;\; .05674)^T$ and $\hat{x} = (1.235 \;\; .05128)^T$, then $\| \hat{x} - x \|_\infty / \| x \|_\infty \approx .0043 \approx 10^{-3}$. Note than $\hat{x}_1$ has about three significant digits that are correct while only one significant digit in $\hat{x}_2$ is correct.

2.2.4 Convergence

We say that a sequence $\{x^{(k)}\}$ of n-vectors *converges* to x if

$$\lim_{k \to \infty} \| x^{(k)} - x \| = 0\,.$$

Note that because of (2.2.4), convergence in the α-norm implies convergence in the β-norm and vice versa.

Problems

P2.2.1 Show that if $x \in \mathbb{R}^n$, then $\lim_{p \to \infty} \| x \|_p = \| x \|_\infty$.

P2.2.2 Prove the Cauchy-Schwartz inequality (2.2.3) by considering the inequality $0 \leq (ax + by)^T(ax + by)$ for suitable scalars a and b .

P2.2.3 Verify that $\| \cdot \|_1$, $\| \cdot \|_2$, and $\| \cdot \|_\infty$ are vector norms.

P2.2.4 Verify (2.2.5)-(2.2.7). When is equality achieved in each result?

P2.2.5 Show that in $\mathbb{R}^n$, $x^{(i)} \to x$ if and only if $x_k^{(i)} \to x_k$ for $k = 1{:}n$.

P2.2.6 Show that any vector norm on $\mathbb{R}^n$ is uniformly continuous by verifying the inequality $|\, \| x \| - \| y \| \,| \leq \| x - y \|$.

P2.2.7 Let $\| \cdot \|$ be a vector norm on $\mathbb{R}^m$ and assume $A \in \mathbb{R}^{m \times n}$. Show that if $\text{rank}(A) = n$, then $\| x \|_A = \| Ax \|$ is a vector norm on $\mathbb{R}^n$.

P2.2.8 Let x and y be in $\mathbb{R}^n$ and define $\psi{:}\mathbb{R} \to \mathbb{R}$ by $\psi(\alpha) = \| x - \alpha y \|_2$. Show that ψ is minimized when $\alpha = x^T y / y^T y$.

P2.2.9 (a) Verify that $\| x \|_p = (|x_1|^p + \cdots + |x_n|^p)^{\frac{1}{p}}$ is a vector norm on $\mathbb{C}^n$. (b) Show that if $x \in \mathbb{C}^n$ then $\| x \|_p \;\leq\; c\left(\| \text{Re}(x) \|_p + \| \text{Im}(x) \|_p\right)$. (c) Find a constant c_n such that $c_n\, (\| \text{Re}(x) \|_2 + \| \text{Im}(x) \|_2) \leq \| x \|_2$ for all $x \in \mathbb{C}^n$.

P2.2.10 Prove or disprove:

$$v \in \mathbb{R}^n \;\Rightarrow\; \| v \|_1 \| v \|_\infty \leq \frac{1 + \sqrt{n}}{2} \| v \|_2.$$

Notes and References for Sec. 2.2

Although a vector norm is "just" a generalization of the absolute value concept, there are some noteworthy subtleties:

J.D. Pryce (1984). "A New Measure of Relative Error for Vectors," *SIAM J. Num. Anal.* *21*, 202-21.

2.3 Matrix Norms

The analysis of matrix algorithms frequently requires use of matrix norms. For example, the quality of a linear system solver may be poor if the matrix of coefficients is "nearly singular." To quantify the notion of near-singularity we need a measure of distance on the space of matrices. Matrix norms provide that measure.

2.3.1 Definitions

Since $\mathbb{R}^{m\times n}$ is isomorphic to $\mathbb{R}^{mn}$, the definition of a matrix norm should be equivalent to the definition of a vector norm. In particular, $f:\mathbb{R}^{m\times n} \rightarrow \mathbb{R}$ is a matrix norm if the following three properties hold:

$$\begin{array}{lll} f(A) \geq 0 & A \in \mathbb{R}^{m\times n}, & (f(A) = 0 \text{ iff } A = 0) \\ f(A+B) \leq f(A) + f(B) & A, B \in \mathbb{R}^{m\times n}, & \\ f(\alpha A) = |\alpha| f(A) & \alpha \in \mathbb{R}, A \in \mathbb{R}^{m\times n}. & \end{array}$$

As with vector norms, we use a double bar notation with subscripts to designate matrix norms, i.e., $\| A \| = f(A)$.

The most frequently used matrix norms in numerical linear algebra are the Frobenius norm,

$$\| A \|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2} \tag{2.3.1}$$

and the p-norms

$$\| A \|_p = \sup_{x\neq 0} \frac{\| Ax \|_p}{\| x \|_p}. \tag{2.3.2}$$

Note that the matrix p-norms are defined in terms of the vector p-norms that we discussed in the previous section. The verification that (2.3.1) and (2.3.2) are matrix norms is left as an exercise. It is clear that $\| A \|_p$ is the p-norm of the largest vector obtained by applying A to a unit p-norm vector:

$$\| A \|_p = \sup_{x\neq 0} \left\| A\left(\frac{x}{\| x \|_p}\right)\right\|_p = \max_{\| x \|_p = 1} \| Ax \|_p .$$

It is important to understand that (2.3.1) and (2.3.2) define families of norms—the 2-norm on $\mathbb{R}^{3\times 2}$ is a different function from the 2-norm on $\mathbb{R}^{5\times 6}$. Thus, the easily verified inequality

$$\| AB \|_p \leq \| A \|_p \| B \|_p \qquad A \in \mathbb{R}^{m\times n}, B \in \mathbb{R}^{n\times q} \tag{2.3.3}$$

is really an observation about the relationship between three different norms. Formally, we say that norms f_1, f_2, and f_3 on $\mathbb{R}^{m\times q}$, $\mathbb{R}^{m\times n}$, and $\mathbb{R}^{n\times q}$ are *mutually consistent* if for all $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times q}$ we have $f_1(AB) \leq f_2(A)f_3(B)$.

Not all matrix norms satisfy the submultiplicative property

$$\| AB \| \leq \| A \| \| B \| . \tag{2.3.4}$$

For example, if $\| A \|_\Delta = \max |a_{ij}|$ and

$$A = B = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix},$$

then $\| AB \|_\Delta > \| A \|_\Delta \| B \|_\Delta$. For the most part we work with norms that satisfy (2.3.4).

The p-norms have the important property that for every $A \in \mathbb{R}^{m\times n}$ and $x \in \mathbb{R}^n$ we have $\| Ax \|_p \leq \| A \|_p \| x \|_p$. More generally, for any vector norm $\| \cdot \|_\alpha$ on $\mathbb{R}^n$ and $\| \cdot \|_\beta$ on $\mathbb{R}^m$ we have $\| Ax \|_\beta \leq \| A \|_{\alpha,\beta} \| x \|_\alpha$ where $\| A \|_{\alpha,\beta}$ is a matrix norm defined by

$$\| A \|_{\alpha,\beta} = \sup_{x\neq 0} \frac{\| Ax \|_\beta}{\| x \|_\alpha}. \tag{2.3.5}$$

We say that $\| \cdot \|_{\alpha,\beta}$ is *subordinate* to the vector norms $\| \cdot \|_\alpha$ and $\| \cdot \|_\beta$. Since the set $\{x \in \mathbb{R}^n : \| x \|_\alpha = 1\}$ is compact and $\| \cdot \|_\beta$ is continuous, it follows that

$$\| A \|_{\alpha,\beta} = \max_{\| x \|_\alpha = 1} \| Ax \|_\beta = \| Ax^* \|_\beta \tag{2.3.6}$$

for some $x^* \in \mathbb{R}^n$ having unit α-norm.

2.3.2 Some Matrix Norm Properties

The Frobenius and p-norms (especially $p = 1, 2, \infty$) satisfy certain inequalities that are frequently used in the analysis of matrix computations. For $A \in \mathbb{R}^{m\times n}$ we have

$$\| A \|_2 \leq \| A \|_F \leq \sqrt{n} \| A \|_2 \tag{2.3.7}$$

$$\max_{i,j} |a_{ij}| \leq \| A \|_2 \leq \sqrt{mn} \max_{i,j} |a_{ij}| \tag{2.3.8}$$

$$\| A \|_1 = \max_{1\leq j\leq n} \sum_{i=1}^{m} |a_{ij}| \tag{2.3.9}$$

$$\| A \|_\infty = \max_{1\leq i\leq m} \sum_{j=1}^{n} |a_{ij}| \tag{2.3.10}$$

$$\frac{1}{\sqrt{n}} \| A \|_\infty \leq \| A \|_2 \leq \sqrt{m} \| A \|_\infty \tag{2.3.11}$$

$$\frac{1}{\sqrt{m}} \| A \|_1 \leq \| A \|_2 \leq \sqrt{n} \| A \|_1 \tag{2.3.12}$$

If $A \in \mathbb{R}^{m\times n}$, $1 \le i_1 \le i_2 \le m$, and $1 \le j_1 \le j_2 \le n$, then

$$\| A(i_1{:}i_2, j_1{:}j_2) \|_p \le \| A \|_p \tag{2.3.13}$$

The proofs of these relations are not hard and are left as exercises.

A sequence $\{A^{(k)}\} \in \mathbb{R}^{m\times n}$ *converges* if $\lim_{k\to\infty} \| A^{(k)} - A \| = 0$. Choice of norm is irrelevant since all norms on $\mathbb{R}^{m\times n}$ are equivalent.

2.3.3 The Matrix 2-Norm

A nice feature of the matrix 1-norm and the matrix ∞-norm is that they are easily computed from (2.3.9) and (2.3.10). A characterization of the 2-norm is considerably more complicated.

Theorem 2.3.1 *If $A \in \mathbb{R}^{m\times n}$, then there exists a unit 2-norm n-vector z such that $A^TAz = \mu^2 z$ where $\mu = \| A \|_2$.*

Proof. Suppose $z \in \mathbb{R}^n$ is a unit vector such that $\| Az \|_2 = \| A \|_2$. Since z maximizes the function

$$g(x) = \frac{1}{2}\frac{\| Ax \|_2^2}{\| x \|_2^2} = \frac{1}{2}\frac{x^TA^TAx}{x^Tx}$$

it follows that it satisfies $\nabla g(z) = 0$ where ∇g is the gradient of g. But a tedious differentiation shows that for $i = 1{:}n$

$$\frac{\partial g(z)}{\partial z_i} = \left[(z^Tz)\sum_{j=1}^{n}(A^TA)_{ij}z_j - (z^TA^TAz)z_i \right] \Big/ (z^Tz)^2 .$$

In vector notation this says $A^TAz = (z^TA^TAz)z$. The theorem follows by setting $\mu = \| Az \|_2$. □

The theorem implies that $\| A \|_2^2$ is a zero of the polynomial $p(\lambda) = \det(A^TA - \lambda I)$. In particular, the 2-norm of A is the square root of the largest eigenvalue of A^TA. We have much more to say about eigenvalues in Chapters 7 and 8. For now, we merely observe that 2-norm computation is iterative and decidedly more complicated than the computation of the matrix 1-norm or ∞-norm. Fortunately, if the object is to obtain an order-of-magnitude estimate of $\| A \|_2$, then (2.3.7), (2.3.11), or (2.3.12) can be used.

As another example of "norm analysis," here is a handy result for 2-norm estimation.

Corollary 2.3.2 *If $A \in \mathbb{R}^{m\times n}$, then $\| A \|_2 \le \sqrt{\| A \|_1 \| A \|_\infty}$.*

Proof. If $z \ne 0$ is such that $A^TAz = \mu^2 z$ with $\mu = \| A \|_2$, then $\mu^2 \| z \|_1 = \| A^TAz \|_1 \le \| A^T \|_1 \| A \|_1 \| z \|_1 = \| A \|_\infty \| A \|_1 \| z \|_1$. □

2.3.4 Perturbations and the Inverse

We frequently use norms to quantify the effect of perturbations or to prove that a sequence of matrices converges to a specified limit. As an illustration of these norm applications, let us quantify the change in A^{-1} as a function of change in A.

Lemma 2.3.3 *If $F \in \mathbb{R}^{n\times n}$ and $\| F \|_p < 1$, then $I - F$ is nonsingular and*

$$(I - F)^{-1} = \sum_{k=0}^{\infty} F^k$$

with

$$\| (I - F)^{-1} \|_p \le \frac{1}{1 - \| F \|_p}.$$

Proof. Suppose $I - F$ is singular. It follows that $(I - F)x = 0$ for some nonzero x. But then $\| x \|_p = \| Fx \|_p$ implies $\| F \|_p \ge 1$, a contradiction. Thus, $I - F$ is nonsingular. To obtain an expression for its inverse consider the identity

$$\left(\sum_{k=0}^{N} F^k \right)(I - F) = I - F^{N+1}.$$

Since $\| F \|_p < 1$ it follows that $\lim_{k\to\infty} F^k = 0$ because $\| F^k \|_p \le \| F \|_p^k$. Thus,

$$\left(\lim_{N\to\infty} \sum_{k=0}^{N} F^k \right)(I - F) = I .$$

It follows that $(I - F)^{-1} = \lim_{N\to\infty} \sum_{k=0}^{N} F^k$. From this it is easy to show that

$$\| (I - F)^{-1} \|_p \le \sum_{k=0}^{\infty} \| F \|_p^k = \frac{1}{1 - \| F \|_p}. \ \square$$

Note that $\| (I - F)^{-1} - I \|_p \le \| F \|_p / (1 - \| F \|_p)$ as a consequence of the lemma. Thus, if $\epsilon \ll 1$, then $O(\epsilon)$ perturbations in I induce $O(\epsilon)$ perturbations in the inverse. We next extend this result to general matrices.

Theorem 2.3.4 *If A is nonsingular and $r \equiv \| A^{-1}E \|_p < 1$, then $A + E$ is nonsingular and $\| (A + E)^{-1} - A^{-1} \|_p \le \| E \|_p \| A^{-1} \|_p^2 / (1 - r)$.*

Proof. Since A is nonsingular $A + E = A(I - F)$ where $F = -A^{-1}E$. Since $\| F \|_p = r < 1$ it follows from Lemma 2.3.3 that $I - F$ is nonsingular and $\| (I - F)^{-1} \|_p < 1/(1 - r)$. Now $(A + E)^{-1} = (I - F)^{-1}A^{-1}$ and so

$$\| (A + E)^{-1} \|_p \le \frac{\| A^{-1} \|_p}{1 - r} .$$

Equation (2.1.3) says that $(A+E)^{-1} - A^{-1} = -A^{-1}E(A+E)^{-1}$ and so by taking norms we find

$$\begin{aligned} \| (A+E)^{-1} - A^{-1} \|_p &\leq \| A^{-1} \|_p \| E \|_p \| (A+E)^{-1} \|_p \\ &\leq \frac{\| A^{-1} \|_p^2 \| E \|_p}{1-r}. \square \end{aligned}$$

Problems

P2.3.1 Show $\| AB \|_p \leq \| A \|_p \| B \|_p$ where $1 \leq p \leq \infty$.

P2.3.2 Let B be any submatrix of A. Show that $\| B \|_p \leq \| A \|_p$.

P2.3.3 Show that if $D = \text{diag}(\mu_1, \ldots, \mu_k) \in \mathbb{R}^{m \times n}$ with $k = \min\{m, n\}$, then $\| D \|_p = \max |\mu_i|$.

P2.3.4 Verify (2.3.7) and (2.3.8).

P2.3.5 Verify (2.3.9) and (2.3.10).

P2.3.6 Verify (2.3.11) and (2.3.12).

P2.3.7 Verify (2.3.13).

P2.3.8 Show that if $0 \neq s \in \mathbb{R}^n$ and $E \in \mathbb{R}^{n \times n}$, then

$$\left\| E \left(I - \frac{ss^T}{s^T s} \right) \right\|_F^2 = \| E \|_F^2 - \frac{\| Es \|_2^2}{s^T s}.$$

P2.3.9 Suppose $u \in \mathbb{R}^m$ and $v \in \mathbb{R}^n$. Show that if $E = uv^T$ then $\| E \|_F = \| E \|_2 = \| u \|_2 \| v \|_2$ and that $\| E \|_\infty \leq \| u \|_\infty \| v \|_1$.

P2.3.10 Suppose $A \in \mathbb{R}^{m \times n}$, $y \in \mathbb{R}^m$, and $0 \neq s \in \mathbb{R}^n$. Show that $E = (y - As)s^T / s^T s$ has the smallest 2-norm of all m-by-n matrices E that satisfy $(A+E)s = y$.

Notes and References for Sec. 2.3

For deeper issues concerning matrix/vector norms, see

F.L. Bauer and C.T. Fike (1960). "Norms and Exclusion Theorems," *Numer. Math. 2*, 137-44.

L. Mirsky (1960). "Symmetric Gauge Functions and Unitarily Invariant Norms," *Quart. J. Math. 11*, 50-59.

A.S. Householder (1964). *The Theory of Matrices in Numerical Analysis*, Dover Publications, New York.

N.J. Higham (1992). "Estimating the Matrix p-Norm," *Numer. Math. 62*, 539-556.

2.4 Finite Precision Matrix Computations

In part, rounding errors are what makes the matrix computation area so nontrivial and interesting. In this section we set up a model of floating point arithmetic and then use it to develop error bounds for floating point dot products, saxpy's, matrix-vector products and matrix-matrix products. For a more comprehensive treatment than what we offer, see Higham (1996) or Wilkinson (1965). The coverage in Forsythe and Moler (1967) and Stewart (1973) is also excellent.

2.4.1 The Floating Point Numbers

When calculations are performed on a computer, each arithmetic operation is generally affected by *roundoff error*. This error arises because the machine hardware can only represent a subset of the real numbers. We denote this subset by $\mathbf{F}$ and refer to its elements as *floating point numbers*. Following conventions set forth in Forsythe, Malcolm, and Moler (1977, pp. 10-29), the floating point number system on a particular computer is characterized by four integers: the *base* β, the *precision* t, and the *exponent range* $[L, U]$. In particular, $\mathbf{F}$ consists of all numbers f of the form

$$f = \pm .d_1 d_2 \ldots d_t \times \beta^e \qquad 0 \leq d_i < \beta, \quad d_1 \neq 0, \quad L \leq e \leq U$$

together with zero. Notice that for a nonzero $f \in \mathbf{F}$ we have $m \leq |f| \leq M$ where

$$m = \beta^{L-1} \quad \text{and} \quad M = \beta^U(1 - \beta^{-t}). \tag{2.4.1}$$

As an example, if $\beta = 2$, $t = 3$, $L = 0$, and $U = 2$, then the non-negative elements of $\mathbf{F}$ are represented by hash marks on the axis displayed in FIG. 2.4.1. Notice that the floating point numbers are not equally spaced. A

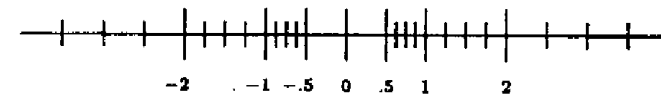


FIGURE 2.4.1 *Sample Floating Point Number System*

typical value for (β, t, L, U) might be (2, 56, -64, 64).

2.4.2 A Model of Floating Point Arithmetic

To make general pronouncements about the effect of rounding errors on a given algorithm, it is necessary to have a model of computer arithmetic on $\mathbf{F}$. To this end define the set G by

$$G = \{ x \in \mathbb{R} : m \leq |x| \leq M \} \cup \{0\} \tag{2.4.2}$$

and the operator fl: $G \to \mathbb{F}$ by

$$fl(x) = \left\{ \begin{array}{l} \text{nearest } c \in \mathbb{F} \text{ to } x \text{ with ties handled} \\ \text{by rounding away from zero.} \end{array} \right\}$$

The fl operator can be shown to satisfy

$$fl(x) = x(1+\epsilon) \qquad |\epsilon| \leq \mathbf{u} \tag{2.4.3}$$

where u is the *unit roundoff* defined by

$$\mathbf{u} = \frac{1}{2}\beta^{1-t}. \tag{2.4.4}$$

Let a and b be any two floating point numbers and let "op" denote any of the four arithmetic operations $+$, $-$, $\times$, $\div$. *If a op $b \in G$, then in our model of floating point arithmetic we assume that the computed version of (a op b) is given by $fl(a \text{ op } b)$.* It follows that $fl(a \text{ op } b) = (a \text{ op } b)(1+\epsilon)$ with $|\epsilon| \leq \mathbf{u}$. Thus,

$$\frac{|fl(a \text{ op } b) - (a \text{ op } b)|}{|a \text{ op } b|} \leq \mathbf{u} \qquad a \text{ op } b \neq 0 \tag{2.4.5}$$

showing that there is small relative error associated with individual arithmetic operations[1]. It is important to realize, however, that this is not necessarily the case when a sequence of operations is involved.

Example 2.4.1 If $\beta = 10$, $t = 3$ floating point arithmetic is used, then it can be shown that $fl[fl(10^{-4}+1) - 1] = 0$ implying a relative error of 1. On the other hand the exact answer is given by $fl[fl(10^{-4} + fl(1-1)] = 10^{-4}$. Floating point arithmetic is not always associative.

If a op $b \notin G$, then an *arithmetic exception* occurs. *Overflow* and *underflow* results whenever $|a \text{ op } b| > M$ or $0 < |a \text{ op } b| < m$ respectively. The handling of these and other exceptions is hardware/system dependent.

2.4.3 Cancellation

Another important aspect of finite precision arithmetic is the phenomenon of *catastrophic cancellation*. Roughly speaking, this term refers to the extreme loss of correct significant digits when small numbers are additively computed from large numbers. A well-known example taken from Forsythe, Malcolm and Moler (1977, pp. 14-16) is the computation of e^{-a} via Taylor series with $a > 0$. The roundoff error associated with this method is

[1] There are important examples of machines whose additive floating point operations satisfy $fl(a \pm b) = (1+\epsilon_1)a \pm (1+\epsilon_2)b$ where $|\epsilon_1|, |\epsilon_2| \leq \mathbf{u}$. In such an environment, the inequality $|fl(a \pm b) - (a \pm b)| \leq \mathbf{u}|a \pm b|$ need not hold.

approximately u times the largest partial sum. For large a, this error can actually be larger than the exact exponential and there will be no correct digits in the answer no matter how many terms in the series are summed. On the other hand, if enough terms in the Taylor series for e^a are added and the result reciprocated, then an estimate of e^{-a} to full precision is attained.

2.4.4 The Absolute Value Notation

Before we proceed with the roundoff analysis of some basic matrix calculations, we acquire some useful notation. Suppose $A \in \mathbb{R}^{m \times n}$ and that we wish to quantify the errors associated with its floating point representation. Denoting the stored version of A by $fl(A)$, we see that

$$[fl(A)]_{ij} = fl(a_{ij}) = a_{ij}(1+\epsilon_{ij}) \qquad |\epsilon_{ij}| \leq \mathbf{u} \tag{2.4.6}$$

for all i and j. A better way to say the same thing results if we adopt two conventions. If A and B are in $\mathbb{R}^{m \times n}$, then

$$B = |A| \quad \Rightarrow \quad b_{ij} = |a_{ij}|, \; i = 1{:}m, \; j = 1{:}n$$

$$B \leq A \quad \Rightarrow \quad b_{ij} \leq a_{ij}, \; i = 1{:}m, \; j = 1{:}n\,.$$

With this notation we see that (2.4.6) has the form

$$|fl(A) - A| \leq \mathbf{u}|A|\,.$$

A relation such as this can be easily turned into a norm inequality, e.g., $\| fl(A) - A \|_1 \leq \mathbf{u} \| A \|_1$. However, when quantifying the rounding errors in a matrix manipulation, the absolute value notation can be a lot more informative because it provides a comment on each (i,j) entry.

2.4.5 Roundoff in Dot Products

We begin our study of finite precision matrix computations by considering the rounding errors that result in the standard dot product algorithm:

$$\begin{array}{l} s = 0 \\ \textbf{for } k = 1{:}n \\ \qquad s = s + x_k y_k \\ \textbf{end} \end{array} \tag{2.4.7}$$

Here, x and y are n-by-1 floating point vectors.

In trying to quantify the rounding errors in this algorithm, we are immediately confronted with a notational problem: the distinction between computed and exact quantities. When the underlying computations are clear, we shall use the $fl(\cdot)$ operator to signify computed quantities.

Thus, $fl(x^Ty)$ denotes the computed output of (2.4.7). Let us bound $|fl(x^Ty) - x^Ty|$. If

$$s_p = fl\left(\sum_{k=1}^{p} x_k y_k\right),$$

then $s_1 = x_1y_1(1+\delta_1)$ with $|\delta_1| \le \mathbf{u}$ and for $p = 2{:}n$

$$\begin{aligned} s_p &= fl(s_{p-1} + fl(x_py_p)) \\ &= (s_{p-1} + x_py_p(1+\delta_p))(1+\epsilon_p) \qquad |\delta_p|, |\epsilon_p| \le \mathbf{u}\,. \end{aligned} \quad (2.4.8)$$

A little algebra shows that

$$fl(x^Ty) = s_n = \sum_{k=1}^{n} x_ky_k(1+\gamma_k)$$

where

$$(1+\gamma_k) = (1+\delta_k)\prod_{j=k}^{n}(1+\epsilon_j)$$

with the convention that $\epsilon_1 = 0$. Thus,

$$|fl(x^Ty) - x^Ty| \quad \le \quad \sum_{k=1}^{n} |x_ky_k||\gamma_k|. \qquad (2.4.9)$$

To proceed further, we must bound the quantities $|\gamma_k|$ in terms of $\mathbf{u}$. The following result is useful for this purpose.

Lemma 2.4.1 *If* $(1+\alpha) = \prod_{k=1}^{n}(1+\alpha_k)$ *where* $|\alpha_k| \le \mathbf{u}$ *and* $n\mathbf{u} \le .01$, *then* $|\alpha| \le 1.01n\mathbf{u}$.

Proof. See Higham (1996, p. 75). □

Applying this result to (2.4.9) under the "reasonable" assumption $n\mathbf{u} \le .01$ gives

$$|fl(x^Ty) - x^Ty| \;\le\; 1.01n\mathbf{u}|x|^T|y|. \qquad (2.4.10)$$

Notice that if $|x^Ty| \ll |x|^T|y|$, then the relative error in $fl(x^Ty)$ may not be small.

2.4.6 Alternative Ways to Quantify Roundoff Error

An easier but less rigorous way of bounding α in Lemma 2.4.1 is to say $|\alpha| \le n\mathbf{u} + O(\mathbf{u}^2)$. With this convention we have

$$|fl(x^Ty) - x^Ty| \le n\mathbf{u}|x|^T|y| + O(\mathbf{u}^2). \qquad (2.4.11)$$

Other ways of expressing the same result include

$$|fl(x^Ty) - x^Ty| \le \phi(n)\mathbf{u}|x|^T|y| \qquad (2.4.12)$$

and

$$|fl(x^Ty) - x^Ty| \le cn\mathbf{u}|x|^T|y|, \qquad (2.4.13)$$

where in (2.4.12) $\phi(n)$ is a "modest" function of n and in (2.4.13) c is a constant of order unity.

We shall not express a preference for any of the error bounding styles shown in (2.4.10)-(2.4.13). This spares us the necessity of translating the roundoff results that appear in the literature into a fixed format. Moreover, paying overly close attention to the details of an error bound is inconsistent with the "philosophy" of roundoff analysis. As Wilkinson (1971, p. 567) says,

> There is still a tendency to attach too much importance to the precise error bounds obtained by an à priori error analysis. In my opinion, the bound itself is usually the least important part of it. The main object of such an analysis is to expose the potential instabilities, if any, of an algorithm so that hopefully from the insight thus obtained one might be led to improved algorithms. Usually the bound itself is weaker than it might have been because of the necessity of restricting the mass of detail to a reasonable level and because of the limitations imposed by expressing the errors in terms of matrix norms. À priori bounds are not, in general, quantities that should be used in practice. Practical error bounds should usually be determined by some form of à posteriori error analysis, since this takes full advantage of the statistical distribution of rounding errors and of any special features, such as sparseness, in the matrix.

It is important to keep these perspectives in mind.

2.4.7 Dot Product Accumulation

Some computers have provision for accumulating dot products in *double precision*. This means that if x and y are floating point vectors with length t mantissas, then the running sum s in (2.4.7) is built up in a register with a $2t$ digit mantissa. Since the multiplication of two t-digit floating point numbers can be stored exactly in a double precision variable, it is only when s is written to single precision memory that any roundoff occurs. In this situation one can usually assert that a computed dot product has good *relative* error, i.e., $fl(x^Ty) = x^Ty(1+\delta)$ where $|\delta| \approx \mathbf{u}$. Thus, the ability to accumulate dot products is very appealing.

2.4.8 Roundoff in Other Basic Matrix Computations

It is easy to show that if A and B are floating point matrices and α is a floating point number, then

$$fl(\alpha A) = \alpha A + E \qquad |E| \le \mathbf{u}|\alpha A| \tag{2.4.14}$$

and

$$fl(A+B) = (A+B) + E \qquad |E| \le \mathbf{u}|A+B|. \tag{2.4.15}$$

As a consequence of these two results, it is easy to verify that computed saxpy's and outer product updates satisfy

$$fl(\alpha x + y) = \alpha x + y + z \qquad |z| \le \mathbf{u}\,(2|\alpha x| + |y|) + O(\mathbf{u}^2) \tag{2.4.16}$$

$$fl(C + uv^T) = C + uv^T + E \qquad |E| \le \mathbf{u}\left(|C| + 2|uv^T|\right) + O(\mathbf{u}^2). \tag{2.4.17}$$

Using (2.4.10) it is easy to show that a dot product based multiplication of two floating point matrices A and B satisfies

$$fl(AB) = AB + E \qquad |E| \le n\mathbf{u}|A||B| + O(\mathbf{u}^2). \tag{2.4.18}$$

The same result applies if a gaxpy or outer product based procedure is used. Notice that matrix multiplication does not necessarily give small relative error since $|AB|$ may be much smaller than $|A||B|$, e.g.,

$$\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ -.99 & 0 \end{bmatrix} = \begin{bmatrix} .01 & 0 \\ 0 & 0 \end{bmatrix}.$$

It is easy to obtain norm bounds from the roundoff results developed thus far. If we look at the 1-norm error in floating point matrix multiplication, then it is easy to show from (2.4.18) that

$$\| fl(AB) - AB \|_1 \;\le\; n\mathbf{u}\| A \|_1 \| B \|_1 + O(\mathbf{u}^2). \tag{2.4.19}$$

2.4.9 Forward and Backward Error Analyses

Each roundoff bound given above is the consequence of a *forward error analysis*. An alternative style of characterizing the roundoff errors in an algorithm is accomplished through a technique known as *backward error analysis*. Here, the rounding errors are related to the data of the problem rather than to its solution. By way of illustration, consider the $n = 2$ version of triangular matrix multiplication. It can be shown that:

$$fl(AB) = \begin{bmatrix} a_{11}b_{11}(1+\epsilon_1) & (a_{11}b_{12}(1+\epsilon_2) + a_{12}b_{22}(1+\epsilon_3))(1+\epsilon_4) \\ 0 & a_{22}b_{22}(1+\epsilon_5) \end{bmatrix}$$

where $|\epsilon_i| \le \mathbf{u}$, for $i = 1{:}5$. However, if we define

$$\hat{A} = \begin{bmatrix} a_{11} & a_{12}(1+\epsilon_3)(1+\epsilon_4) \\ 0 & a_{22}(1+\epsilon_5) \end{bmatrix}$$

and

$$\hat{B} = \begin{bmatrix} b_{11}(1+\epsilon_1) & b_{12}(1+\epsilon_2)(1+\epsilon_4) \\ 0 & b_{22} \end{bmatrix},$$

then it is easily verified that $fl(AB) = \hat{A}\hat{B}$. Moreover,

$$\hat{A} = A + E \qquad |E| \le 2\mathbf{u}|A| + O(\mathbf{u}^2)$$

$$\hat{B} = B + F \qquad |F| \le 2\mathbf{u}|B| + O(\mathbf{u}^2).$$

In other words, the computed product is the exact product of slightly perturbed A and B.

2.4.10 Error in Strassen Multiplication

In §1.3.8 we outlined an unconventional matrix multiplication procedure due to Strassen (1969). It is instructive to compare the effect of roundoff in this method with the effect of roundoff in any of the conventional matrix multiplication methods of §1.1.

It can be shown that the Strassen approach (Algorithm 1.3.1) produces a $\hat{C} = fl(AB)$ that satisfies an inequality of the form (2.4.19). This is perfectly satisfactory in many applications. However, the $\hat{C}$ that Strassen's method produces does *not* always satisfy an inequality of the form (2.4.18). To see this, suppose

$$A \;=\; B \;=\begin{bmatrix} .99 & .0010 \\ .0010 & .99 \end{bmatrix}$$

and that we execute Algorithm 1.3.1 using 2-digit floating point arithmetic. Among other things, the following quantities are computed:

$$\begin{aligned} \hat{P}_3 &= fl(.99(.001 - .99)) = -.98 \\ \hat{P}_5 &= fl((.99 + .001).99) = .98 \\ \hat{c}_{12} &= fl(\hat{P}_3 + \hat{P}_5) = 0.0 \end{aligned}$$

Now in exact arithmetic $c_{12} = 2(.001)(.99) = .00198$ and thus Algorithm 1.3.1 produces a $\hat{c}_{12}$ with no correct significant digits. The Strassen approach gets into trouble in this example because small off-diagonal entries are combined with large diagonal entries. Note that in conventional matrix multiplication neither b_{12} and b_{22} nor a_{11} and a_{12} are summed. Thus the contribution of

the small off-diagonal elements is not lost. Indeed, for the above A and B a conventional matrix multiply gives $\hat{c}_{12} = .0020$.

Failure to produce a componentwise accurate $\hat{C}$ can be a serious shortcoming in some applications. For example, in Markov processes the a_{ij}, b_{ij}, and c_{ij} are transition probabilities and are therefore nonnegative. It may be critical to compute c_{ij} accurately if it reflects a particularly important probability in the modeled phenomena. Note that if $A \geq 0$ and $B \geq 0$, then conventional matrix multiplication produces a product $\hat{C}$ that has small componentwise relative error:

$$|\hat{C} - C| \;\leq\; n\mathbf{u}|A|\,|B| + O(\mathbf{u}^2) \;=\; n\mathbf{u}|C| + O(\mathbf{u}^2)\,.$$

This follows from (2.4.18). Because we cannot say the same for the Strassen approach, we conclude that Algorithm 1.3.1 is not attractive for *certain* nonnegative matrix multiplication problems *if* relatively accurate $\hat{c}_{ij}$ are required.

Extrapolating from this discussion we reach two fairly obvious but important conclusions:

- Different methods for computing the same quantity can produce substantially different results.
- Whether or not an algorithm produces satisfactory results depends upon the type of problem solved and the goals of the user.

These observations are clarified in subsequent chapters and are intimately related to the concepts of algorithm stability and problem condition.

Problems

P2.4.1 Show that if (2.4.7) is applied with $y = x$, then $fl(x^Tx) = x^Tx(1+\alpha)$ where $|\alpha| \leq n\mathbf{u} + O(\mathbf{u}^2)$.

P2.4.2 Prove (2.4.3).

P2.4.3 Show that if $E \in \mathbb{R}^{m\times n}$ with $m \geq n$, then $\|\,|E|\,\|_2 \leq \sqrt{n}\|\,E\,\|_2$. This result is useful when deriving norm bounds from absolute value bounds.

P2.4.4 Assume the existence of a square root function satisfying $fl(\sqrt{x}) = \sqrt{x}(1+\epsilon)$ with $|\epsilon| \leq \mathbf{u}$. Give an algorithm for computing $\|\,x\,\|_2$ and bound the rounding errors.

P2.4.5 Suppose A and B are n-by-n upper triangular floating point matrices. If $\hat{C} = fl(AB)$ is computed using one of the conventional §1.1 algorithms, does it follow that $\hat{C} = \hat{A}\hat{B}$ where $\hat{A}$ and $\hat{B}$ are close to A and B?

P2.4.6 Suppose A and B are n-by-n floating point matrices and that A is nonsingular with $\|\,|A^{-1}|\,|A|\,\|_\infty = \tau$. Show that if $\hat{C} = fl(AB)$ is obtained using any of the algorithms in §1.1, then there exists a $\hat{B}$ so $\hat{C} = A\hat{B}$ and $\|\,\hat{B} - B\,\|_\infty \leq n\mathbf{u}\tau\|\,B\,\|_\infty + O(\mathbf{u}^2)$.

P2.4.7 Prove (2.4.18).

Notes and References for Sec. 2.4

For a general introduction to the effects of roundoff error, we recommend

J.H. Wilkinson (1963). *Rounding Errors in Algebraic Processes*, Prentice-Hall, Englewood Cliffs, NJ.
J.H. Wilkinson (1971). "Modern Error Analysis," *SIAM Review 13*, 548–68.
D. Kahaner, C.B. Moler, and S. Nash (1988). *Numerical Methods and Software*, Prentice-Hall, Englewood Cliffs, NJ.
F. Chaitin-Chatelin and V. Frayssé (1996). *Lectures on Finite Precision Computations*, SIAM Publications, Philadelphia.

More recent developments in error analysis involve interval analysis, the building of statistical models of roundoff error, and the automating of the analysis itself:

T.E. Hull and J.R. Swenson (1966). "Tests of Probabilistic Models for Propagation of Roundoff Errors," *Comm. ACM. 9*, 108–13.
J. Larson and A. Sameh (1978). "Efficient Calculation of the Effects of Roundoff Errors," *ACM Trans. Math. Soft. 4*, 228–36.
W. Miller and D. Spooner (1978). "Software for Roundoff Analysis, II," *ACM Trans. Math. Soft. 4*, 369–90.
J.M. Yohe (1979). "Software for Interval Arithmetic: A Reasonable Portable Package," *ACM Trans. Math. Soft. 5*, 50–63.

Anyone engaged in serious software development needs a thorough understanding of floating point arithmetic. A good way to begin acquiring knowledge in this direction is to read about the IEEE floating point standard in

D. Goldberg (1991). "What Every Computer Scientist Should Know About Floating Point Arithmetic," *ACM Surveys 23*, 5–48.

See also

R.P. Brent (1978). "A Fortran Multiple Precision Arithmetic Package," *ACM Trans. Math. Soft. 4*, 57–70.
R.P. Brent (1978). "Algorithm 524 MP, a Fortran Multiple Precision Arithmetic Package," *ACM Trans. Math. Soft. 4*, 71–81.
J.W. Demmel (1984). "Underflow and the Reliability of Numerical Software," *SIAM J. Sci. and Stat. Comp. 5*, 887–919.
U.W. Kulisch and W.L. Miranker (1986). "The Arithmetic of the Digital Computer," *SIAM Review 28*, 1–40.
W.J. Cody (1988). "ALGORITHM 665 MACHAR: A Subroutine to Dynamically Determine Machine Parameters," *ACM Trans. Math. Soft. 14*, 303–311.
D.H. Bailey, H.D. Simon, J. T. Barton, M.J. Fouts (1989). "Floating Point Arithmetic in Future Supercomputers," *Int'l J. Supercomputing Appl. 3*, 86–90.
D.H. Bailey (1993). "Algorithm 719: Multiprecision Translation and Execution of FORTRAN Programs," *ACM Trans. Math. Soft. 19*, 288–319.

The subtleties associated with the development of high-quality software, even for "simple" problems, are immense. A good example is the design of a subroutine to compute 2-norms

J.M. Blue (1978). "A Portable FORTRAN Program to Find the Euclidean Norm of a Vector," *ACM Trans. Math. Soft. 4*, 15–23.

For an analysis of the Strassen algorithm and other "fast" linear algebra procedures see

R.P. Brent (1970). "Error Analysis of Algorithms for Matrix Multiplication and Triangular Decomposition Using Winograd's Identity," *Numer. Math.* 16, 145–156.
W. Miller (1975). "Computational Complexity and Numerical Stability," *SIAM J. Computing* 4, 97–107.
N.J. Higham (1992). "Stability of a Method for Multiplying Complex Matrices with Three Real Matrix Multiplications," *SIAM J. Matrix Anal. Appl.* 13, 681–687.
J.W. Demmel and N.J. Higham (1992). "Stability of Block Algorithms with Fast Level-3 BLAS," *ACM Trans. Math. Soft.* 18, 274–291.

2.5 Orthogonality and the SVD

Orthogonality has a very prominent role to play in matrix computations. After establishing a few definitions we prove the extremely useful singular value decomposition (SVD). Among other things, the SVD enables us to intelligently handle the matrix rank problem. The concept of rank, though perfectly clear in the exact arithmetic context, is tricky in the presence of roundoff error and fuzzy data. With the SVD we can introduce the practical notion of numerical rank.

2.5.1 Orthogonality

A set of vectors $\{x_1, \ldots, x_p\}$ in $\mathbb{R}^m$ is *orthogonal* if $x_i^T x_j = 0$ whenever $i \neq j$ and *orthonormal* if $x_i^T x_j = \delta_{ij}$. Intuitively, orthogonal vectors are maximally independent for they point in totally different directions.

A collection of subspaces $S_1, \ldots, S_p$ in $\mathbb{R}^m$ is *mutually orthogonal* if $x^T y = 0$ whenever $x \in S_i$ and $y \in S_j$ for $i \neq j$. The *orthogonal complement* of a subspace $S \subseteq \mathbb{R}^m$ is defined by

$$S^\perp = \{y \in \mathbb{R}^m : y^T x = 0 \text{ for all } x \in S\}$$

and it is not hard to show that $\text{ran}(A)^\perp = \text{null}(A^T)$. The vectors $v_1, \ldots, v_k$ form an *orthonormal* basis for a subspace $S \subseteq \mathbb{R}^m$ if they are orthonormal and span S.

A matrix $Q \in \mathbb{R}^{m \times m}$ is said to be *orthogonal* if $Q^T Q = I$. If $Q = [\, q_1, \ldots, q_m \,]$ is orthogonal, then the q_i form an orthonormal basis for $\mathbb{R}^m$. It is always possible to extend such a basis to a full orthonormal basis $\{v_1, \ldots, v_m\}$ for $\mathbb{R}^m$:

Theorem 2.5.1 *If $V_1 \in \mathbb{R}^{n \times r}$ has orthonormal columns, then there exists $V_2 \in \mathbb{R}^{n \times (n-r)}$ such that*

$$V = [\, V_1 \, V_2 \,]$$

is orthogonal. Note that $\text{ran}(V_1)^\perp = \text{ran}(V_2)$.

Proof. This is a standard result from introductory linear algebra. It is also a corollary of the QR factorization that we present in §5.2. □

2.5.2 Norms and Orthogonal Transformations

The 2-norm is invariant under orthogonal transformation, for if $Q^T Q = I$, then $\| Qx \|_2^2 = x^T Q^T Q x = x^T x = \| x \|_2^2$. The matrix 2-norm and the Frobenius norm are also invariant with respect to orthogonal transformations. In particular, it is easy to show that for all orthogonal Q and Z of appropriate dimensions we have

$$\| QAZ \|_F = \| A \|_F \qquad (2.5.1)$$

and

$$\| QAZ \|_2 = \| A \|_2 . \qquad (2.5.2)$$

2.5.3 The Singular Value Decomposition

The theory of norms developed in the previous two sections can be used to prove the extremely useful singular value decomposition.

Theorem 2.5.2 (Singular Value Decomposition (SVD)) *If A is a real m-by-n matrix, then there exist orthogonal matrices*

$$U = [\, u_1, \ldots, u_m \,] \in \mathbb{R}^{m \times m} \quad \textit{and} \quad V = [\, v_1, \ldots, v_n \,] \in \mathbb{R}^{n \times n}$$

such that

$$U^T A V = \text{diag}(\sigma_1, \ldots, \sigma_p) \in \mathbb{R}^{m \times n} \qquad p = \min\{m, n\}$$

where $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_p \geq 0$.

Proof. Let $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$ be unit 2-norm vectors that satisfy $Ax = \sigma y$ with $\sigma = \| A \|_2$. From Theorem 2.5.1 there exist $V_2 \in \mathbb{R}^{n \times (n-1)}$ and $U_2 \in \mathbb{R}^{m \times (m-1)}$ so $V = [\, x \;\; V_2 \,] \in \mathbb{R}^{n \times n}$ and $U = [\, y \;\; U_2 \,] \in \mathbb{R}^{m \times m}$ are orthogonal. It is not hard to show that $U^T A V$ has the following structure:

$$U^T A V = \begin{bmatrix} \sigma & w^T \\ 0 & B \end{bmatrix} \equiv A_1 .$$

Since

$$\left\| A_1 \left(\begin{bmatrix} \sigma \\ w \end{bmatrix} \right) \right\|_2^2 \geq (\sigma^2 + w^T w)^2$$

we have $\| A_1 \|_2^2 \geq (\sigma^2 + w^T w)$. But $\sigma^2 = \| A \|_2^2 = \| A_1 \|_2^2$, and so we must have $w = 0$. An obvious induction argument completes the proof of the theorem. □

The σ_i are the *singular values* of A and the vectors u_i and v_i are the ith *left singular vector* and the ith *right singular vector* respectively. It

is easy to verify by comparing columns in the equations $AV = U\Sigma$ and $A^TU = V\Sigma^T$ that

$$\left.\begin{aligned} Av_i &= \sigma_i u_i \\ A^T u_i &= \sigma_i v_i \end{aligned}\right\} i = 1{:}\min\{m,n\}$$

It is convenient to have the following notation for designating singular values:

$$\begin{aligned} \sigma_i(A) &= \text{the } i\text{th largest singular value of } A, \\ \sigma_{max}(A) &= \text{the largest singular value of } A, \\ \sigma_{min}(A) &= \text{the smallest singular value of } A. \end{aligned}$$

The singular values of a matrix A are precisely the lengths of the semi-axes of the hyperellipsoid E defined by $E = \{ Ax : \| x \|_2 = 1 \}$.

Example 2.5.1

$$A = \begin{bmatrix} .96 & 1.72 \\ 2.28 & .96 \end{bmatrix} = U\Sigma V^T = \begin{bmatrix} .6 & -.8 \\ .8 & .6 \end{bmatrix} \begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} .8 & .6 \\ .6 & -.8 \end{bmatrix}^T .$$

The SVD reveals a great deal about the structure of a matrix. If the SVD of A is given by Theorem 2.5.2, and we define r by

$$\sigma_1 \ge \cdots \ge \sigma_r > \sigma_{r+1} = \cdots = \sigma_p = 0,$$

then

$$\text{rank}(A) = r \tag{2.5.3}$$
$$\text{null}(A) = \text{span}\{v_{r+1}, \ldots, v_n\} \tag{2.5.4}$$
$$\text{ran}(A) = \text{span}\{u_1, \ldots, u_r\}, \tag{2.5.5}$$

and we have the *SVD expansion*

$$A = \sum_{i=1}^{r} \sigma_i u_i v_i^T . \tag{2.5.6}$$

Various 2-norm and Frobenius norm properties have connections to the SVD. If $A \in \mathbb{R}^{m\times n}$, then

$$\| A \|_F^2 = \sigma_1^2 + \cdots + \sigma_p^2 \qquad p = \min\{m,n\} \tag{2.5.7}$$
$$\| A \|_2 = \sigma_1 \tag{2.5.8}$$
$$\min_{x \ne 0} \frac{\| Ax \|_2}{\| x \|_2} = \sigma_n \qquad (m \ge n). \tag{2.5.9}$$

2.5.4 The Thin SVD

If $A = U\Sigma V^T \in \mathbb{R}^{m\times n}$ is the SVD of A and $m \ge n$, then

$$A = U_1 \Sigma_1 V^T$$

where

$$U_1 = U(:,1{:}n) = [\, u_1, \ldots, u_n \,] \in \mathbb{R}^{m\times n}$$

and

$$\Sigma_1 = \Sigma(1{:}n, 1{:}n) = \text{diag}(\sigma_1, \ldots, \sigma_n) \in \mathbb{R}^{n\times n}.$$

We refer to this much-used, trimmed down version of the SVD as the *thin SVD*.

2.5.5 Rank Deficiency and the SVD

One of the most valuable aspects of the SVD is that it enables us to deal sensibly with the concept of matrix rank. Numerous theorems in linear algebra have the form "if such-and-such a matrix has full rank, then such-and-such a property holds." While neat and aesthetic, results of this flavor do not help us address the numerical difficulties frequently encountered in situations where near rank deficiency prevails. Rounding errors and fuzzy data make rank determination a nontrivial exercise. Indeed, for some small ϵ we may be interested in the ϵ-rank of a matrix which we define by

$$\text{rank}(A, \epsilon) = \min_{\| A - B \|_2 \le \epsilon} \text{rank}(B).$$

Thus, if A is obtained in a laboratory with each a_{ij} correct to within $\pm .001$, then it might make sense to look at $\text{rank}(A, .001)$. Along the same lines, if A is an m-by-n floating point matrix then it is reasonable to regard A as *numerically rank deficient* if $\text{rank}(A, \epsilon) < \min\{m,n\}$ with $\epsilon = \mathbf{u} \| A \|_2$.

Numerical rank deficiency and ϵ-rank are nicely characterized in terms of the SVD because the singular values indicate how near a given matrix is to a matrix of lower rank.

Theorem 2.5.3 *Let the SVD of $A \in \mathbb{R}^{m\times n}$ be given by Theorem 2.5.2. If $k < r = \text{rank}(A)$ and*

$$A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T, \tag{2.5.10}$$

then

$$\min_{\text{rank}(B) = k} \| A - B \|_2 = \| A - A_k \|_2 = \sigma_{k+1}. \tag{2.5.11}$$

Proof. Since $U^T A_k V = \mathrm{diag}(\sigma_1, \ldots, \sigma_k, 0, \ldots, 0)$ it follows that $\mathrm{rank}(A_k) = k$ and that $U^T(A - A_k)V = \mathrm{diag}(0, \ldots, 0, \sigma_{k+1}, \ldots, \sigma_p)$ and so $\| A - A_k \|_2 = \sigma_{k+1}$.

Now suppose $\mathrm{rank}(B) = k$ for some $B \in \mathbb{R}^{m \times n}$. It follows that we can find orthonormal vectors $x_1, \ldots, x_{n-k}$ so $\mathrm{null}(B) = \mathrm{span}\{x_1, \ldots, x_{n-k}\}$. A dimension argument shows that

$$\mathrm{span}\{x_1, \ldots, x_{n-k}\} \cap \mathrm{span}\{v_1, \ldots, v_{k+1}\} \neq \{0\}.$$

Let z be a unit 2-norm vector in this intersection. Since $Bz = 0$ and

$$Az = \sum_{i=1}^{k+1} \sigma_i (v_i^T z) u_i$$

we have

$$\| A - B \|_2^2 \geq \| (A - B)z \|_2^2 = \| Az \|_2^2 = \sum_{i=1}^{k+1} \sigma_i^2 (v_i^T z)^2 \geq \sigma_{k+1}^2$$

completing the proof of the theorem. □

Theorem 2.5.3 says that the smallest singular value of A is the 2-norm distance of A to the set of all rank-deficient matrices. It also follows that the set of full rank matrices in $\mathbb{R}^{m \times n}$ is both open and dense.

Finally, if $r_\epsilon = \mathrm{rank}(A, \epsilon)$, then

$$\sigma_1 \geq \cdots \geq \sigma_{r_\epsilon} > \epsilon \geq \sigma_{r_\epsilon + 1} \geq \cdots \geq \sigma_p \qquad p = \min\{m, n\}.$$

We have more to say about the numerical rank issue in §5.5 and §12.2.

2.5.6 Unitary Matrices

Over the complex field the unitary matrices correspond to the orthogonal matrices. In particular, $Q \in \mathbb{C}^{n \times n}$ is *unitary* if $Q^H Q = QQ^H = I_n$. Unitary matrices preserve 2-norm. The SVD of a complex matrix involves unitary matrices. If $A \in \mathbb{C}^{m \times n}$, then there exist unitary matrices $U \in \mathbb{C}^{m \times m}$ and $V \in \mathbb{C}^{n \times n}$ such that

$$U^H AV = \mathrm{diag}(\sigma_1, \ldots, \sigma_p) \in \mathbb{R}^{m \times n} \qquad p = \min\{m, n\}$$

where $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_p \geq 0$.

Problems

P2.5.1 Show that if S is real and $S^T = -S$, then $I - S$ is nonsingular and the matrix $(I - S)^{-1}(I + S)$ is orthogonal. This is known as the *Cayley transform* of S.

P2.5.2 Show that a triangular orthogonal matrix is diagonal.

P2.5.3 Show that if $Q = Q_1 + iQ_2$ is unitary with $Q_1, Q_2 \in \mathbb{R}^{n \times n}$, then the $2n$-by-$2n$ real matrix

$$Z = \begin{bmatrix} Q_1 & -Q_2 \\ Q_2 & Q_1 \end{bmatrix}$$

is orthogonal.

P2.5.4 Establish properties (2.5.3)-(2.5.9).

P2.5.5 Prove that

$$\sigma_{max}(A) = \max_{y \in \mathbb{R}^m, x \in \mathbb{R}^n} \frac{y^T Ax}{\| x \|_2 \| y \|_2}$$

P2.5.6 For the 2-by-2 matrix $A = \begin{bmatrix} w & x \\ y & z \end{bmatrix}$, derive expressions for $\sigma_{max}(A)$ and $\sigma_{min}(A)$ that are functions of w, x, y, and z.

P2.5.7 Show that any matrix in $\mathbb{R}^{m \times n}$ is the limit of a sequence of full rank matrices.

P2.5.8 Show that if $A \in \mathbb{R}^{m \times n}$ has rank n, then $\| A(A^T A)^{-1} A^T \|_2 = 1$.

P2.5.9 What is the nearest rank-one matrix to $A = \begin{bmatrix} 1 & M \\ 0 & 1 \end{bmatrix}$ in the Frobenius norm?

P2.5.10 Show that if $A \in \mathbb{R}^{m \times n}$ then $\| A \|_F \leq \sqrt{\mathrm{rank}(A)} \, \| A \|_2$, thereby sharpening (2.3.7).

Notes and References for Sec. 2.5

Forsythe and Moler (1967) offer a good account of the SVD's role in the analysis of the $Ax = b$ problem. Their proof of the decomposition is more traditional than ours in that it makes use of the eigenvalue theory for symmetric matrices. Historical SVD references include

E. Beltrami (1873). "Sulle Funzioni Bilineari," *Gionale di Mathematiche 11*, 98–106.
C. Eckart and G. Young (1939). "A Principal Axis Transformation for Non-Hermitian Matrices," *Bull. Amer. Math. Soc. 45*, 118–21.
G.W. Stewart (1993). "On the Early History of the Singular Value Decomposition," *SIAM Review 35*, 551–566.

One of the most significant developments in scientific computation has been the increased use of the SVD in application areas that require the intelligent handling of matrix rank. The range of applications is impressive. One of the most interesting is

C.B. Moler and D. Morrison (1983). "Singular Value Analysis of Cryptograms," *Amer. Math. Monthly 90*, 78–87.

For generalizations of the SVD to infinite dimensional Hilbert space, see

I.C. Gohberg and M.G. Krein (1969). *Introduction to the Theory of Linear Non-Self Adjoint Operators* , Amer. Math. Soc., Providence, R.I.
F. Smithies (1970). *Integral Equations*, Cambridge University Press, Cambridge.

Reducing the rank of a matrix as in Theorem 2.5.3 when the perturbing matrix is constrained is discussed in

J.W. Demmel (1987). "The smallest perturbation of a submatrix which lowers the rank and constrained total least squares problems, *SIAM J. Numer. Anal. 24*, 199–206.

G.H. Golub, A. Hoffman, and G.W. Stewart (1988). "A Generalization of the Eckart-Young-Mirsky Approximation Theorem." *Lin. Alg. and Its Applic.* *88/89*, 317–328.
G.A. Watson (1988). "The Smallest Perturbation of a Submatrix which Lowers the Rank of the Matrix," *IMA J. Numer. Anal.* *8*, 295–304.

2.6 Projections and the CS Decomposition

If the object of a computation is to compute a matrix or a vector, then norms are useful for assessing the accuracy of the answer or for measuring progress during an iteration. If the object of a computation is to compute a subspace, then to make similar comments we need to be able to quantify the distance between two subspaces. Orthogonal projections are critical in this regard. After the elementary concepts are established we discuss the CS decomposition. This is an SVD-like decomposition that is handy when having to compare a pair of subspaces. We begin with the notion of an orthogonal projection.

2.6.1 Orthogonal Projections

Let $S \subseteq \mathbb{R}^n$ be a subspace. $P \in \mathbb{R}^{n\times n}$ is the *orthogonal projection* onto S if $\text{ran}(P) = S$, $P^2 = P$, and $P^T = P$. From this definition it is easy to show that if $x \in \mathbb{R}^n$, then $Px \in S$ and $(I-P)x \in S^\perp$.

If P_1 and P_2 are each orthogonal projections, then for any $z \in \mathbb{R}^n$ we have

$$\| (P_1 - P_2)z \|_2^2 \;=\; (P_1 z)^T(I-P_2)z + (P_2 z)^T(I-P_1)z.$$

If $\text{ran}(P_1) = \text{ran}(P_2) = S$, then the right-hand side of this expression is zero showing that the orthogonal projection for a subspace is unique. If the columns of $V = [\, v_1, \ldots, v_k \,]$ are an orthonormal basis for a subspace S, then it is easy to show that $P = VV^T$ is the unique orthogonal projection onto S. Note that if $v \in \mathbb{R}^n$, then $P = vv^T/v^Tv$ is the orthogonal projection onto $S = \text{span}\{v\}$.

2.6.2 SVD-Related Projections

There are several important orthogonal projections associated with the singular value decomposition. Suppose $A = U\Sigma V^T \in \mathbb{R}^{m\times n}$ is the SVD of A and that $r = \text{rank}(A)$. If we have the U and V partitionings

$$U = [\, \underset{r}{U_r} \quad \underset{m-r}{\tilde{U}_r} \,] \qquad V = [\, \underset{r}{V_r} \quad \underset{n-r}{\tilde{V}_r} \,]$$

then

$$\begin{aligned}
V_rV_r^T &= \text{projection on to null}(A)^\perp = \text{ran}(A^T)\\
\tilde{V}_r\tilde{V}_r^T &= \text{projection on to null}(A)\\
U_rU_r^T &= \text{projection on to ran}(A)\\
\tilde{U}_r\tilde{U}_r^T &= \text{projection on to ran}(A)^\perp = \text{null}(A^T)
\end{aligned}$$

2.6.3 Distance Between Subspaces

The one-to-one correspondence between subspaces and orthogonal projections enables us to devise a notion of distance between subspaces. Suppose S_1 and S_2 are subspaces of $\mathbb{R}^n$ and that $\dim(S_1) = \dim(S_2)$. We define the *distance* between these two spaces by

$$\text{dist}(S_1, S_2) \;=\; \| P_1 - P_2 \|_2 \tag{2.6.1}$$

where P_i is the orthogonal projection onto S_i. The distance between a pair of subspaces can be characterized in terms of the blocks of a certain orthogonal matrix.

Theorem 2.6.1 *Suppose*

$$W = [\, \underset{k}{W_1} \quad \underset{n-k}{W_2} \,] \qquad Z = [\, \underset{k}{Z_1} \quad \underset{n-k}{Z_2} \,]$$

are n-by-n orthogonal matrices. If $S_1 = \text{ran}(W_1)$ *and* $S_2 = \text{ran}(Z_1)$*, then*

$$\text{dist}(S_1, S_2) \;=\; \| W_1^T Z_2 \|_2 \;=\; \| Z_1^T W_2 \|_2.$$

Proof.

$$\begin{aligned}
\text{dist}(S_1,S_2) &= \| W_1W_1^T - Z_1Z_1^T \|_2 = \| W^T(W_1W_1^T - Z_1Z_1^T)Z \|_2\\
&= \left\| \begin{bmatrix} 0 & W_1^TZ_2 \\ -W_2^TZ_1 & 0 \end{bmatrix} \right\|_2 .
\end{aligned}$$

Note that the matrices $W_2^TZ_1$ and $W_1^TZ_2$ are submatrices of the orthogonal matrix

$$Q = \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} \equiv \begin{bmatrix} W_1^TZ_1 & W_1^TZ_2 \\ W_2^TZ_1 & W_2^TZ_2 \end{bmatrix} = W^TZ.$$

Our goal is to show that $\| Q_{21} \|_2 = \| Q_{12} \|_2$. Since Q is orthogonal it follows from

$$Q \begin{bmatrix} x \\ 0 \end{bmatrix} = \begin{bmatrix} Q_{11}x \\ Q_{21}x \end{bmatrix}$$

that

$$1 = \| Q_{11}x \|_2^2 + \| Q_{21}x \|_2^2$$

for all unit 2-norm $x \in \mathbb{R}^k$. Thus,

$$\begin{aligned}
\| Q_{21} \|_2^2 &= \max_{\| x \|_2=1} \| Q_{21}x \|_2^2 = 1 - \min_{\| x \|_2=1} \| Q_{11}x \|_2^2\\
&= 1 - \sigma_{min}(Q_{11})^2.
\end{aligned}$$

Analogously, by working with Q^T (which is also orthogonal) it is possible to show that

$$\| Q_{12}^T \|_2^2 = 1 - \sigma_{min}(Q_{11}^T)^2.$$

and therefore

$$\| Q_{12} \|_2^2 = 1 - \sigma_{min}(Q_{11})^2.$$

Thus, $\| Q_{21} \|_2 = \| Q_{12} \|_2$. □

Note that if S_1 and S_2 are subspaces in $\mathbb{R}^n$ with the same dimension, then

$$0 \le \text{dist}(S_1, S_2) \le 1.$$

The distance is zero if $S_1 = S_2$ and one if $S_1 \bigcap S_2^\perp \neq \{0\}$.

A more refined analysis of the blocks of the Q matrix above sheds more light on the difference between a pair of subspaces. This requires a special SVD-like decomposition for orthogonal matrices.

2.6.4 The CS Decomposition

The blocks of an orthogonal matrix partitioned into 2-by-2 form have highly related SVDs. This is the gist of the *CS decomposition*. We prove a very useful special case first.

Theorem 2.6.2 (The CS Decomposition (Thin Version)) *Consider the matrix*

$$Q = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} \qquad Q_1 \in \mathbb{R}^{m_1 \times n},\ Q_2 \in \mathbb{R}^{m_2 \times n}$$

where $m_1 \ge n$ and $m_2 \ge n$. If the columns of Q are orthonormal, then there exist orthogonal matrices $U_1 \in \mathbb{R}^{m_1 \times m_1}$, $U_2 \in \mathbb{R}^{m_2 \times m_2}$, and $V_1 \in \mathbb{R}^{n \times n}$ such that

$$\begin{bmatrix} U_1 & 0 \\ 0 & U_2 \end{bmatrix}^T \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} V_1 = \begin{bmatrix} C \\ S \end{bmatrix}$$

where

$$\begin{aligned} C &= \text{diag}(\cos(\theta_1), \ldots, \cos(\theta_n)), \\ S &= \text{diag}(\sin(\theta_1), \ldots, \sin(\theta_n)), \end{aligned}$$

and

$$0 \le \theta_1 \le \theta_2 \le \cdots \le \theta_n \le \frac{\Pi}{2}.$$

Proof. Since $\| Q_{11} \|_2 \le \| Q \|_2 = 1$, the singular values of Q_{11} are all in the interval $[0, 1]$. Let

$$U_1^T Q_1 V_1 = C = \text{diag}(c_1, \ldots, c_n) = \underset{\displaystyle t \quad n-t}{\begin{bmatrix} I_t & 0 \\ 0 & \Sigma \end{bmatrix}} \begin{matrix} t \\ m_1 - t \end{matrix}$$

be the SVD of Q_1 where we assume

$$1 = c_1 = \cdots = c_t > c_{t+1} \ge \cdots \ge c_n \ge 0.$$

To complete the proof of the theorem we must construct the orthogonal matrix U_2. If

$$Q_2 V_1 = \underset{\displaystyle t \qquad n-t}{[\ W_1 \quad W_2\ ]},$$

then

$$\begin{bmatrix} U_1 & 0 \\ 0 & I_{m_2} \end{bmatrix}^T \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} V_1 = \begin{bmatrix} I_t & 0 \\ 0 & \Sigma \\ W_1 & W_2 \end{bmatrix}.$$

Since the columns of this matrix have unit 2-norm, $W_1 = 0$. The columns of W_2 are nonzero and mutually orthogonal because

$$W_2^T W_2 = I_{n-t} - \Sigma^T \Sigma \equiv \text{diag}(1 - c_{t+1}^2, \ldots, 1 - c_n^2)$$

is nonsingular. If $s_k = \sqrt{1 - c_k^2}$ for $k = 1{:}n$, then the columns of

$$Z = W_2\, \text{diag}(1/s_{t+1}, \ldots, 1/s_n)$$

are orthonormal. By Theorem 2.5.1 there exists an orthogonal matrix $U_2 \in \mathbb{R}^{m_2 \times m_2}$ with $U_2(:, t+1{:}n) = Z$. It is easy to verify that

$$U_2^T Q_2 V_1 = \text{diag}(s_1, \ldots, s_n) \equiv S.$$

Since $c_k^2 + s_k^2 = 1$ for $k = 1{:}n$, it follows that these quantities are the required cosines and sines. □

Using the same sort of techniques it is possible to prove the following more general version of the decomposition:

Theorem 2.6.3 (CS Decomposition (General Version)) *If*

$$Q = \left[\begin{array}{c|c} Q_{11} & Q_{12} \\ \hline Q_{21} & Q_{22} \end{array}\right]$$

is a 2-by-2 (arbitrary) partitioning of an n-by-n orthogonal matrix, then there exist orthogonal

$$U = \left[\begin{array}{c|c} U_1 & 0 \\ \hline 0 & U_2 \end{array}\right] \quad \text{and} \quad V = \left[\begin{array}{c|c} V_1 & 0 \\ \hline 0 & V_2 \end{array}\right]$$

such that

$$U^T Q V = \left[\begin{array}{ccc|ccc} I & 0 & 0 & 0 & 0 & 0 \\ 0 & C & 0 & 0 & S & 0 \\ 0 & 0 & 0 & 0 & 0 & I \\ \hline 0 & 0 & 0 & I & 0 & 0 \\ 0 & S & 0 & 0 & -C & 0 \\ 0 & 0 & I & 0 & 0 & 0 \end{array}\right]$$

where $C = \text{diag}(c_1, \ldots, c_p)$ *and* $S = \text{diag}(s_1, \ldots, s_p)$ *are square diagonal matrices with* $0 < c_i, s_i < 1$.

Proof. See Paige and Saunders (1981) for details. We have suppressed the dimensions of the zero submatrices, some of which may be empty. □

The essential message of the decomposition is that the SVDs of the Q_{ij} are highly related.

Example 2.6.1 The matrix

$$Q = \left[\begin{array}{rr|rrr} -0.7576 & 0.3697 & 0.3838 & 0.2126 & -0.3112 \\ -0.4077 & -0.1552 & -0.1129 & 0.2676 & 0.8517 \\ -0.0488 & 0.7240 & -0.6730 & -0.1301 & 0.0602 \\ \hline -0.2287 & 0.0088 & 0.2235 & -0.9235 & 0.2120 \\ 0.4530 & 0.5612 & 0.5806 & 0.1162 & 0.3595 \end{array}\right]$$

is orthogonal and with the indicated partitioning can be reduced to

$$U^TQV = \left[\begin{array}{rr|rrr} 0.9837 & 0.0000 & 0.1800 & 0.0000 & 0.0000 \\ 0.0000 & 0.6781 & 0.0000 & 0.7349 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 1.0000 \\ \hline 0.1800 & 0.0000 & -0.9837 & 0.0000 & 0.0000 \\ 0.0000 & 0.7349 & 0.0000 & -0.6781 & 0.0000 \end{array}\right]$$

The angles associated with the cosines and sines turn out to be very important in a number of applications. See §12.4.

Problems

P2.6.1 Show that if P is an orthogonal projection, then $Q = I - 2P$ is orthogonal.

P2.6.2 What are the singular values of an orthogonal projection?

P2.6.3 Suppose $S_1 = \text{span}\{x\}$ and $S_2 = \text{span}\{y\}$, where x and y are unit 2-norm vectors in $\mathbb{R}^2$. Working only with the definition of $\text{dist}(\cdot,\cdot)$, show that $\text{dist}(S_1, S_2) = \sqrt{1-(x^Ty)^2}$ verifying that the distance between S_1 and S_2 equals the sine of the angle between x and y.

Notes and References for Sec. 2.6

The following papers discuss various aspects of the CS decomposition:

C. Davis and W. Kahan (1970). "The Rotation of Eigenvectors by a Perturbation III," *SIAM J. Num. Anal.* 7, 1–46.

G.W. Stewart (1977). "On the Perturbation of Pseudo-Inverses, Projections and Linear Least Squares Problems," *SIAM Review 19*, 634–662.

C.C. Paige and M. Saunders (1981). "Toward a Generalized Singular Value Decomposition," *SIAM J. Num. Anal. 18*, 398–405.

C.C. Paige and M. Wei (1994). "History and Generality of the CS Decomposition," *Lin. Alg. and Its Applic. 208/209*, 303–326.

See §8.7 for some computational details.

For a deeper geometrical understanding of the CS decomposition and the notion of distance between subspaces, see

T.A. Arias, A. Edelman, and S. Smith (1996). "Conjugate Gradient and Newton's Method on the Grassman and Stiefel Manifolds," to appear in *SIAM J. Matrix Anal. Appl.*

2.7 The Sensitivity of Square Systems

We now use some of the tools developed in previous sections to analyze the linear system problem $Ax = b$ where $A \in \mathbb{R}^{n\times n}$ is nonsingular and $b \in \mathbb{R}^n$. Our aim is to examine how perturbations in A and b affect the solution x. A much more detailed treatment may be found in Higham (1996).

2.7.1 An SVD Analysis

If

$$A = \sum_{i=1}^{n} \sigma_i u_i v_i^T = U\Sigma V^T$$

is the SVD of A, then

$$x = A^{-1}b = (U\Sigma V^T)^{-1}b = \sum_{i=1}^{n} \frac{u_i^T b}{\sigma_i} v_i. \qquad (2.7.1)$$

This expansion shows that small changes in A or b can induce relatively large changes in x if σ_n is small.

It should come as no surprise that the magnitude of σ_n should have a bearing on the sensitivity of the $Ax = b$ problem when we recall from Theorem 2.5.3 that σ_n is the distance from A to the set of singular matrices. As the matrix of coefficients approaches this set, it is intuitively clear that the solution x should be increasingly sensitive to perturbations.

2.7.2 Condition

A precise measure of linear system sensitivity can be obtained by considering the parameterized system

$$(A + \epsilon F)x(\epsilon) = b + \epsilon f \qquad x(0) = x$$

where $F \in \mathbb{R}^{n\times n}$ and $f \in \mathbb{R}^n$. If A is nonsingular, then it is clear that $x(\epsilon)$ is differentiable in a neighborhood of zero. Moreover, $\dot{x}(0) = A^{-1}(f - Fx)$ and thus, the Taylor series expansion for $x(\epsilon)$ has the form

$$x(\epsilon) = x + \epsilon\dot{x}(0) + O(\epsilon^2).$$

Using any vector norm and consistent matrix norm we obtain

$$\frac{\| x(\epsilon) - x \|}{\| x \|} \leq |\epsilon| \, \| A^{-1} \| \left\{ \frac{\| f \|}{\| x \|} + \| F \| \right\} + O(\epsilon^2). \tag{2.7.2}$$

For square matrices A define the *condition number* $\kappa(A)$ by

$$\kappa(A) = \| A \| \, \| A^{-1} \| \tag{2.7.3}$$

with the convention that $\kappa(A) = \infty$ for singular A. Using the inequality $\| b \| \leq \| A \| \, \| x \|$ it follows from (2.7.2) that

$$\frac{\| x(\epsilon) - x \|}{\| x \|} \leq \kappa(A)(\rho_A + \rho_b) + O(\epsilon^2) \tag{2.7.4}$$

where

$$\rho_A = |\epsilon| \frac{\| F \|}{\| A \|} \quad \text{and} \quad \rho_b = |\epsilon| \frac{\| f \|}{\| b \|}$$

represent the relative errors in A and b, respectively. Thus, the relative error in x can be $\kappa(A)$ times the relative error in A and b. In this sense, the condition number $\kappa(A)$ quantifies the sensitivity of the $Ax = b$ problem.

Note that $\kappa(\cdot)$ depends on the underlying norm and subscripts are used accordingly, e.g.,

$$\kappa_2(A) = \| A \|_2 \| A^{-1} \|_2 = \frac{\sigma_1(A)}{\sigma_n(A)}. \tag{2.7.5}$$

Thus, the 2-norm condition of a matrix A measures the elongation of the hyperellipsoid $\{Ax : \| x \|_2 = 1\}$.

We mention two other characterizations of the condition number. For p-norm condition numbers, we have

$$\frac{1}{\kappa_p(A)} = \min_{A + \Delta A \text{ singular}} \frac{\| \Delta A \|_p}{\| A \|_p}. \tag{2.7.6}$$

This result may be found in Kahan (1966) and shows that $\kappa_p(A)$ measures the relative p-norm distance from A to the set of singular matrices.

For any norm, we also have

$$\kappa(A) = \lim_{\epsilon \to 0} \sup_{\| \Delta A \| \leq \epsilon \| A \|} \frac{\| (A + \Delta A)^{-1} - A^{-1} \|}{\epsilon} \frac{1}{\| A^{-1} \|}. \tag{2.7.7}$$

This imposing result merely says that the condition number is a normalized Frechet derivative of the map $A \to A^{-1}$. Further details may be found in Rice (1966b). Recall that we were initially led to $\kappa(A)$ through differentiation.

If $\kappa(A)$ is large, then A is said to be an *ill-conditioned* matrix. Note that this is a norm-dependent property[2]. However, any two condition numbers $\kappa_\alpha(\cdot)$ and $\kappa_\beta(\cdot)$ on $\mathbb{R}^{n \times n}$ are equivalent in that constants c_1 and c_2 can be found for which

$$c_1 \kappa_\alpha(A) \leq \kappa_\beta(A) \leq c_2 \kappa_\alpha(A) \qquad A \in \mathbb{R}^{n \times n}.$$

For example, on $\mathbb{R}^{n \times n}$ we have

$$\begin{aligned} \frac{1}{n} \kappa_2(A) &\leq \kappa_1(A) \leq n\kappa_2(A) \\ \frac{1}{n} \kappa_\infty(A) &\leq \kappa_2(A) \leq n\kappa_\infty(A) \\ \frac{1}{n^2} \kappa_1(A) &\leq \kappa_\infty(A) \leq n^2 \kappa_1(A). \end{aligned} \tag{2.7.8}$$

Thus, if a matrix is ill-conditioned in the α-norm, it is ill-conditioned in the β-norm modulo the constants c_1 and c_2 above.

For any of the p-norms, we have $\kappa_p(A) \geq 1$. Matrices with small condition numbers are said to be *well-conditioned* . In the 2-norm, orthogonal matrices are perfectly conditioned in that $\kappa_2(Q) = 1$ if Q is orthogonal.

2.7.3 Determinants and Nearness to Singularity

It is natural to consider how well determinant size measures ill-conditioning. If $\det(A) = 0$ is equivalent to singularity, is $\det(A) \approx 0$ equivalent to near singularity? Unfortunately, there is little correlation between $\det(A)$ and the condition of $Ax = b$. For example, the matrix B_n defined by

$$B_n = \begin{bmatrix} 1 & -1 & \cdots & -1 \\ 0 & 1 & \cdots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \in \mathbb{R}^{n \times n} \tag{2.7.9}$$

has determinant 1, but $\kappa_\infty(B_n) = n2^{n-1}$. On the other hand, a very well conditioned matrix can have a very small determinant. For example,

$$D_n = \text{diag}(10^{-1}, \ldots, 10^{-1}) \in \mathbb{R}^{n \times n}$$

satisfies $\kappa_p(D_n) = 1$ although $\det(D_n) = 10^{-n}$.

2.7.4 A Rigorous Norm Bound

Recall that the derivation of (2.7.4) was valuable because it highlighted the connection between $\kappa(A)$ and the rate of change of $x(\epsilon)$ at $\epsilon = 0$. However,

[2] It also depends upon the definition of "large." The matter is pursued in §3.5

it is a little unsatisfying because it is contingent on ϵ being "small enough" and because it sheds no light on the size of the $O(\epsilon^2)$ term. In this and the next subsection we develop some additional $Ax = b$ perturbation theorems that are completely rigorous.

We first establish a useful lemma that indicates in terms of $\kappa(A)$ when we can expect a perturbed system to be nonsingular.

Lemma 2.7.1 *Suppose*

$$\begin{aligned} Ax &= b & A \in \mathbb{R}^{n\times n},\ 0 \neq b \in \mathbb{R}^n \\ (A+\Delta A)y &= b + \Delta b & \Delta A \in \mathbb{R}^{n\times n},\ \Delta b \in \mathbb{R}^n \end{aligned}$$

with $\| \Delta A \| \le \epsilon \| A \|$ *and* $\| \Delta b \| \le \epsilon \| b \|$. *If* $\epsilon\kappa(A) = r < 1$, *then* $A + \Delta A$ *is nonsingular and*

$$\frac{\| y \|}{\| x \|} \le \frac{1+r}{1-r}.$$

Proof. Since $\| A^{-1}\Delta A \| \le \epsilon \| A^{-1} \| \, \| A \| = r < 1$ it follows from Theorem 2.3.4 that $(A + \Delta A)$ is nonsingular. Using Lemma 2.3.3 and the equality $(I + A^{-1}\Delta A)y = x + A^{-1}\Delta b$ we find

$$\begin{aligned} \| y \| &\le \| (I + A^{-1}\Delta A)^{-1} \| \left(\| x \| + \epsilon \| A^{-1} \| \, \| b \| \right) \\ &\le \frac{1}{1-r} \left(\| x \| + \epsilon \| A^{-1} \| \, \| b \| \right) = \frac{1}{1-r}\left(\| x \| + r \frac{\| b \|}{\| A \|} \right). \end{aligned}$$

Since $\| b \| = \| Ax \| \le \| A \| \, \| x \|$ it follows that

$$\| y \| \le \frac{1}{1-r} \left(\| x \| + r \| x \| \right). \ \square$$

We are now set to establish a rigorous $Ax = b$ perturbation bound.

Theorem 2.7.2 *If the conditions of Lemma 2.7.1 hold, then*

$$\frac{\| y - x \|}{\| x \|} \le \frac{2\epsilon}{1-r}\kappa(A) \tag{2.7.10}$$

Proof. Since

$$y - x = A^{-1}\Delta b - A^{-1}\Delta A y \tag{2.7.11}$$

we have $\| y - x \| \le \epsilon \| A^{-1} \| \, \| b \| + \epsilon \| A^{-1} \| \, \| A \| \, \| y \|$ and so

$$\begin{aligned} \frac{\| y - x \|}{\| x \|} &\le \epsilon\kappa(A) \frac{\| b \|}{\| A \| \, \| x \|} + \epsilon\kappa(A) \frac{\| y \|}{\| x \|} \\ &\le \epsilon\kappa(A)\left(1 + \frac{1+r}{1-r}\right) = \frac{2\epsilon}{1-r}\kappa(A). \ \square \end{aligned}$$

Example 2.7.1 The $Ax = b$ problem

$$\begin{bmatrix} 1 & 0 \\ 0 & 10^{-6} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 10^{-6} \end{bmatrix}$$

has solution $x = (1, 1)^T$ and condition $\kappa_\infty(A) = 10^6$. If $\Delta b = (10^{-6}, 0)^T$, $\Delta A = 0$, and $(A + \Delta A)y = b + \Delta b$, then $y = (1 + 10^{-6}, 1)^T$ and the inequality (2.7.10) says

$$10^{-6} = \frac{\| x - y \|_\infty}{\| x \|_\infty} \ll \frac{\| \Delta b \|_\infty}{\| b \|_\infty}\kappa_\infty(A) = 10^{-6}10^6 = 1.$$

Thus, the upper bound in (2.7.10) can be a gross overestimate of the error induced by the perturbation. On the other hand, if $\Delta b = (0, 10^{-6})^T$, $\Delta A = 0$, and $(A+\Delta A)y = b+\Delta b$, then this inequality says

$$\frac{10^0}{10^0} \le 2 \times 10^{-6}10^6.$$

Thus, there are perturbations for which the bound in (2.7.10) is essentially attained.

2.7.5 Some Rigorous Componentwise Bounds

We conclude this section by showing that a more refined perturbation theory is possible if componentwise perturbation bounds are in effect and if we make use of the absolute value notation.

Theorem 2.7.3 *Suppose*

$$\begin{aligned} Ax &= b & A \in \mathbb{R}^{n\times n},\ 0 \neq b \in \mathbb{R}^n \\ (A+\Delta A)y &= b + \Delta b & \Delta A \in \mathbb{R}^{n\times n},\ \Delta b \in \mathbb{R}^n \end{aligned}$$

and that $|\Delta A| \le \epsilon |A|$ *and* $|\Delta b| \le \epsilon |b|$. *If* $\delta\kappa_\infty(A) = r < 1$, *then* $(A + \Delta A)$ *is nonsingular and*

$$\frac{\| y - x \|_\infty}{\| x \|_\infty} \le \frac{2\delta}{1-r} \| \, |A^{-1}| \, |A| \, \|_\infty.$$

Proof. Since $\| \Delta A \|_\infty \le \epsilon \| A \|_\infty$ and $\| \Delta b \|_\infty \le \epsilon \| b \|_\infty$ the conditions of Lemma 2.7.1 are satisfied in the infinity norm. This implies that $A + \Delta A$ is nonsingular and

$$\frac{\| y \|_\infty}{\| x \|_\infty} \le \frac{1+r}{1-r}.$$

Now using (2.7.11) we find

$$\begin{aligned} |y - x| &\le |A^{-1}| \, |\Delta b| + |A^{-1}| \, |\Delta A| \, |y| \\ &\le \epsilon |A^{-1}| \, |b| + \epsilon |A^{-1}| \, |A| \, |y| \le \epsilon |A^{-1}| \, |A| \left(|x| + |y| \right). \end{aligned}$$

If we take norms, then

$$\| y - x \|_\infty \le \epsilon \| \, |A^{-1}| \, |A| \, \|_\infty \left(\| x \|_\infty + \frac{1+r}{1-r} \| x \|_\infty \right).$$

The theorem follows upon division by $\| x \|_\infty$. □

We refer to the quantity $\| |A^{-1}| |A| \|_\infty$ as the *Skeel condition number*. It has been effectively used in the analysis of several important linear system computations. See §3.5.

Lastly, we report on the results of Oettli and Prager (1964) that indicate when an approximate solution $\hat{x} \in \mathbb{R}^n$ to the n-by-n system $Ax = b$ satisfies a perturbed system with prescribed structure. In particular, suppose $E \in \mathbb{R}^{n\times n}$ and $f \in \mathbb{R}^n$ are given and have nonnegative entries. We seek $\Delta A \in \mathbb{R}^{n\times n}$, $\Delta b \in \mathbb{R}^n$, and $\omega \geq 0$ such that

$$(A + \Delta A)\hat{x} = b + \Delta b \qquad |\Delta A| \leq \omega E\,, \ |\Delta b| \leq \omega f\,. \tag{2.7.12}$$

Note that by properly choosing E and f the perturbed system can take on certain qualities. For example, if $E = |A|$ and $f = |b|$ and ω is small, then $\hat{x}$ satisfies a nearby system in the componentwise sense. Oettli and Prager (1964) show that for a given A, b, $\hat{x}$, E, and f the smallest ω possible in (2.7.12) is given by

$$\omega_{min} = \max_{1\leq i\leq n} \frac{|A\hat{x} - b|_i}{(E|\hat{x}| + f)_i}\,.$$

If $A\hat{x} = b$ then $\omega_{min} = 0$. On the other hand, if $\omega_{min} = \infty$, then $\hat{x}$ does not satisfy any system of the prescribed perturbation structure.

Problems

P2.7.1 Show that if $\| I \| \geq 1$, then $\kappa(A) \geq 1$.

P2.7.2 Show that for a given norm, $\kappa(AB) \leq \kappa(A)\kappa(B)$ and that $\kappa(\alpha A) = \kappa(A)$ for all nonzero α.

P2.7.3 Relate the 2-norm condition of $X \in \mathbb{R}^{m\times n}$ $(m \geq n)$ to the 2-norm condition of the matrices

$$B = \begin{bmatrix} I_m & X \\ 0 & I_n \end{bmatrix}$$

and

$$C = \begin{bmatrix} X \\ I_n \end{bmatrix}.$$

Notes and References for Sec. 2.7

The condition concept is thoroughly investigated in

J. Rice (1966). "A Theory of Condition," *SIAM J. Num. Anal. 3*, 287–310.
W. Kahan (1966). "Numerical Linear Algebra," *Canadian Math. Bull. 9*, 757–801.

References for componentwise perturbation theory include

W. Oettli and W. Prager (1964). "Compatibility of Approximate Solutions of Linear Equations with Given Error Bounds for Coefficients and Right Hand Sides," *Numer. Math. 6*, 405–409.
J.E. Cope and B.W. Rust (1979). "Bounds on solutions of systems with accurate data," *SIAM J. Num. Anal. 16*, 950–63.
R.D. Skeel (1979). "Scaling for numerical stability in Gaussian Elimination," *J. ACM 26*, 494–526.
J.W. Demmel (1992). "The Componentwise Distance to the Nearest Singular Matrix," *SIAM J. Matrix Anal. Appl. 13*, 10–19.
D.J. Higham and N.J. Higham (1992). "Componentwise Perturbation Theory for Linear Systems with Multiple Right-Hand Sides," *Lin. Alg. and Its Applic. 174*, 111–129.
N.J. Higham (1994). "A Survey of Componentwise Perturbation Theory in Numerical Linear Algebra," in *Mathematics of Computation 1943–1993: A Half Century of Computational Mathematics*, W. Gautschi (ed.), Volume 48 of *Proceedings of Symposia in Applied Mathematics*, American Mathematical Society, Providence, Rhode Island.
S. Chandrasekaren and I.C.F. Ipsen (1995). "On the Sensitivity of Solution Components in Linear Systems of Equations," *SIAM J. Matrix Anal. Appl. 16*, 93–112.

The reciprocal of the condition number measures how near a given $Ax = b$ problem is to singularity. The importance of knowing how near a given problem is to a difficult or insoluble problem has come to be appreciated in many computational settings. See

A. Laub(1985). "Numerical Linear Algebra Aspects of Control Design Computations," *IEEE Trans. Auto. Cont. AC-30*, 97–108.
J. L. Barlow (1986). "On the Smallest Positive Singular Value of an M-Matrix with Applications to Ergodic Markov Chains," *SIAM J. Alg. and Disc. Struct. 7*, 414–424.
J.W. Demmel (1987). "On the Distance to the Nearest Ill-Posed Problem," *Numer. Math. 51*, 251–289.
J.W. Demmel (1988). "The Probability that a Numerical Analysis Problem is Difficult," *Math. Comp. 50*, 449–480.
N.J. Higham (1989). "Matrix Nearness Problems and Applications," in *Applications of Matrix Theory*, M.J.C. Gover and S. Barnett (eds), Oxford University Press, Oxford UK, 1–27.

Chapter 3

General Linear Systems

§3.1 Triangular Systems
§3.2 The LU Factorization
§3.3 Roundoff Analysis of Gaussian Elimination
§3.4 Pivoting
§3.5 Improving and Estimating Accuracy

The problem of solving a linear system $Ax = b$ is central in scientific computation. In this chapter we focus on the method of Gaussian elimination, the algorithm of choice when A is square, dense, and unstructured. When A does not fall into this category, then the algorithms of Chapters 4, 5, and 10 are of interest. Some parallel $Ax = b$ solvers are discussed in Chapter 6.

We motivate the method of Gaussian elimination in §3.1 by discussing the ease with which triangular systems can be solved. The conversion of a general system to triangular form via Gauss transformations is then presented in §3.2 where the "language" of matrix factorizations is introduced. Unfortunately, the derived method behaves very poorly on a nontrivial class of problems. Our error analysis in §3.3 pinpoints the difficulty and motivates §3.4, where the concept of pivoting is introduced. In the final section we comment upon the important practical issues associated with scaling, iterative improvement, and condition estimation.

Before You Begin

Chapter 1, §§2.1-2.5, and §2.7 are assumed. Complementary references include Forsythe and Moler (1967), Stewart (1973), Hager (1988), Watkins (1991), Ciarlet (1992), Datta (1995), Higham (1996), Trefethen and Bau (1996), and Demmel (1996). Some MATLAB functions important to this chapter are lu, cond, rcond, and the "backslash" operator "\". LAPACK connections include

LAPACK: Triangular Systems	
_TRSV	Solves $Ax = b$
_TRSM	Solves $AX = B$
_TRCON	Condition estimate
_TRRFS	Solve $AX = B$, $A^TX = B$ with error bounds
_TRTRS	Solve $AX = B$, $A^TX = B$
_TRTRI	A^{-1}

LAPACK: General Linear Systems	
_GESV	Solve $AX = B$
_GECON	Condition estimate via $PA = LU$
_GERFS	Improve $AX = B$, $A^TX = B$, $A^HX = B$ solutions with error bounds
_GESVX	Solve $AX = B$, $A^TX = B$, $A^HX = B$ with condition estimate
_GETRF	$PA = LU$
_GETRS	Solve $AX = B$, $A^TX = B$, $A^HX = B$ via $PA = LU$
_GETRI	A^{-1}
_GEEQU	Equilibration

3.1 Triangular Systems

Traditional factorization methods for linear systems involve the conversion of the given square system to a triangular system that has the same solution. This section is about the solution of triangular systems.

3.1.1 Forward Substitution

Consider the following 2-by-2 lower triangular system:

$$\begin{bmatrix} \ell_{11} & 0 \\ \ell_{21} & \ell_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}.$$

If $\ell_{11}\ell_{22} \neq 0$, then the unknowns can be determined sequentially:

$$\begin{aligned} x_1 &= b_1/\ell_{11} \\ x_2 &= (b_2 - \ell_{21}x_1)/\ell_{22}. \end{aligned}$$

This is the 2-by-2 version of an algorithm known as *forward substitution*. The general procedure is obtained by solving the ith equation in $Lx = b$ for x_i:

$$x_i = \left(b_i - \sum_{j=1}^{i-1} \ell_{ij}x_j \right) \Big/ \ell_{ii}$$

If this is evaluated for $i = 1{:}n$, then a complete specification of x is obtained. Note that at the ith stage the dot product of $L(i, 1{:}i-1)$ and $x(1{:}i-1)$ is required. Since b_i only is involved in the formula for x_i, the former may be overwritten by the latter:

Algorithm 3.1.1 (Forward Substitution: Row Version) If $L \in \mathbb{R}^{n\times n}$ is lower triangular and $b \in \mathbb{R}^n$, then this algorithm overwrites b with the solution to $Lx = b$. L is assumed to be nonsingular.

```
b(1) = b(1)/L(1,1)
for i = 2:n
      b(i) = (b(i) - L(i,1:i-1)b(1:i-1))/L(i,i)
end
```

This algorithm requires n^2 flops. Note that L is accessed by row. The computed solution $\hat{x}$ satisfies:

$$(L+F)\hat{x} = b \qquad |F| \le n\mathbf{u}|L| + O(\mathbf{u}^2) \tag{3.1.1}$$

For a proof, see Higham (1996). It says that the computed solution exactly satisfies a slightly perturbed system. Moreover, each entry in the perturbing matrix F is small relative to the corresponding element of L.

3.1.2 Back Substitution

The analogous algorithm for upper triangular systems $Ux = b$ is called *back-substitution*. The recipe for x_i is prescribed by

$$x_i = \left(b_i - \sum_{j=i+1}^{n} u_{ij}x_j \right) \Big/ u_{ii}$$

and once again b_i can be overwritten by x_i.

Algorithm 3.1.2 (Back Substitution: Row Version) If $U \in \mathbb{R}^{n\times n}$ is upper triangular and $b \in \mathbb{R}^n$, then the following algorithm overwrites b with the solution to $Ux = b$. U is assumed to be nonsingular.

```
b(n) = b(n)/U(n,n)
for i = n-1: -1:1
      b(i) = (b(i) - U(i,i+1:n)b(i+1:n))/U(i,i)
end
```

This algorithm requires n^2 flops and accesses U by row. The computed solution $\hat{x}$ obtained by the algorithm can be shown to satisfy

$$(U+F)\hat{x} = b \qquad |F| \le n\mathbf{u}|U| + O(\mathbf{u}^2). \tag{3.1.2}$$

3.1.3 Column Oriented Versions

Column oriented versions of the above procedures can be obtained by reversing loop orders. To understand what this means from the algebraic point of view, consider forward substitution. Once x_1 is resolved, it can be removed from equations 2 through n and we proceed with the reduced system $L(2{:}n, 2{:}n)x(2{:}n) = b(2{:}n) - x(1)L(2{:}n, 1)$. We then compute x_2 and remove it from equations 3 through n, etc. Thus, if this approach is applied to

$$\begin{bmatrix} 2 & 0 & 0 \\ 1 & 5 & 0 \\ 7 & 9 & 8 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6 \\ 2 \\ 5 \end{bmatrix}$$

we find $x_1 = 3$ and then deal with the 2-by-2 system

$$\begin{bmatrix} 5 & 0 \\ 9 & 8 \end{bmatrix} \begin{bmatrix} x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 5 \end{bmatrix} - 3\begin{bmatrix} 1 \\ 7 \end{bmatrix} = \begin{bmatrix} -1 \\ -16 \end{bmatrix}.$$

Here is the complete procedure with overwriting.

Algorithm 3.1.3 (Forward Substitution: Column Version) If $L \in \mathbb{R}^{n\times n}$ is lower triangular and $b \in \mathbb{R}^n$, then this algorithm overwrites b with the solution to $Lx = b$. L is assumed to be nonsingular.

```
for j = 1:n-1
      b(j) = b(j)/L(j,j)
      b(j+1:n) = b(j+1:n) - b(j)L(j+1:n,j)
end
b(n) = b(n)/L(n,n)
```

It is also possible to obtain a column-oriented saxpy procedure for back-substitution.

Algorithm 3.1.4 (Back Substitution: Column Version) If $U \in \mathbb{R}^{n\times n}$ is upper triangular and $b \in \mathbb{R}^n$, then this algorithm overwrites b with the solution to $Ux = b$. U is assumed to be nonsingular.

```
for j = n: -1:2
      b(j) = b(j)/U(j,j)
      b(1:j-1) = b(1:j-1) - b(j)U(1:j-1,j)
end
b(1) = b(1)/U(1,1)
```

Note that the dominant operation in both Algorithms 3.1.3 and 3.1.4 is the saxpy operation. The roundoff behavior of these saxpy implementations is essentially the same as for the dot product versions.

The accuracy of a computed solution to a triangular system is often surprisingly good. See Higham (1996).

3.1.4 Multiple Right Hand Sides

Consider the problem of computing a solution $X \in \mathbb{R}^{n\times q}$ to $LX = B$ where $L \in \mathbb{R}^{n\times n}$ is lower triangular and $B \in \mathbb{R}^{n\times q}$. This is the *multiple right hand side* forward substitution problem. We show that such a problem can be solved by a block algorithm that is rich in matrix multiplication assuming that q and n are large enough. This turns out to be important in subsequent sections where various block factorization schemes are discussed. We mention that although we are considering here just the lower triangular problem, everything we say applies to the upper triangular case as well.

To develop a block forward substitution algorithm we partition the equation $LX = B$ as follows:

$$\begin{bmatrix} L_{11} & 0 & \cdots & 0 \\ L_{21} & L_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ L_{N1} & L_{N2} & \cdots & L_{NN} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_N \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_N \end{bmatrix}. \qquad (3.1.3)$$

Assume that the diagonal blocks are square. Paralleling the development of Algorithm 3.1.3, we solve the system $L_{11}X_1 = B_1$ for X_1 and then remove X_1 from block equations 2 through N:

$$\begin{bmatrix} L_{22} & 0 & \cdots & 0 \\ L_{32} & L_{33} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ L_{N2} & L_{N3} & \cdots & L_{NN} \end{bmatrix} \begin{bmatrix} X_2 \\ X_3 \\ \vdots \\ X_N \end{bmatrix} = \begin{bmatrix} B_2 - L_{21}X_1 \\ B_3 - L_{31}X_1 \\ \vdots \\ B_N - L_{N1}X_1 \end{bmatrix}.$$

Continuing in this way we obtain the following block saxpy forward elimination scheme:

```
for j = 1:N
      Solve L_jj X_j = B_j
      for i = j + 1:N                                    (3.1.4)
            B_i = B_i - L_ij X_j
      end
end
```

Notice that the i-loop oversees a single block saxpy update of the form

$$\begin{bmatrix} B_{j+1} \\ \vdots \\ B_N \end{bmatrix} = \begin{bmatrix} B_{j+1} \\ \vdots \\ B_N \end{bmatrix} - \begin{bmatrix} L_{j+1,j} \\ \vdots \\ L_{N,j} \end{bmatrix} X_j .$$

For this to be handled as a matrix multiplication in a given architecture it is clear that the blocking in (3.1.3) must give sufficiently "big" X_j. Let us assume that this is the case if each X_j has at least r rows. This can be accomplished if $N = \text{ceil}(n/r)$ and $X_1, \ldots, X_{N-1} \in \mathbb{R}^{r\times q}$ and $X_N \in \mathbb{R}^{(n-(N-1)r)\times q}$.

3.1.5 The Level-3 Fraction

It is handy to adopt a measure that quantifies the amount of matrix multiplication in a given algorithm. To this end we define the *level-3 fraction* of an algorithm to be the fraction of flops that occur in the context of matrix multiplication. We call such flops *level-3 flops*.

Let us determine the level-3 fraction for (3.1.4) with the simplifying assumption that $n = rN$. (The same conclusions hold with the unequal blocking described above.) Because there are N applications of r-by-r forward elimination (the level-2 portion of the computation) and n^2 flops overall, the level-3 fraction is approximately given by

$$1 - \frac{Nr^2}{n^2} = 1 - \frac{1}{N}$$

Thus, for large N almost all flops are level-3 flops and it makes sense to choose N as large as possible subject to the constraint that the underlying architecture can achieve a high level of performance when processing block saxpy's of width at least $r = n/N$.

3.1.6 Non-square Triangular System Solving

The problem of solving nonsquare, m-by-n triangular systems deserves some mention. Consider first the lower triangular case when $m \geq n$, i.e.,

$$\begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} x = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \qquad \begin{matrix} L_{11} \in \mathbb{R}^{n\times n} & b_1 \in \mathbb{R}^n \\ L_{21} \in \mathbb{R}^{(m-n)\times n} & b_2 \in \mathbb{R}^{m-n} \end{matrix}$$

Assume that L_{11} is lower triangular, and nonsingular. If we apply forward elimination to $L_{11}x = b_1$ then x solves the system provided $L_{21}(L_{11}^{-1}b_1) = b_2$. Otherwise, there is no solution to the overall system. In such a case least squares minimization may be appropriate. See Chapter 5.

Now consider the lower triangular system $Lx = b$ when the number of columns n exceeds the number of rows m. In this case apply forward substitution to the square system $L(1{:}m, 1{:}m)x(1{:}m, 1{:}m) = b$ and prescribe an arbitrary value for $x(m + 1{:}n)$. See §5.7 for additional comments on systems that have more unknowns than equations.

The handling of nonsquare upper triangular systems is similar. Details are left to the reader.

3.1.7 Unit Triangular Systems

A *unit* triangular matrix is a triangular matrix with ones on the diagonal. Many of the triangular matrix computations that follow have this added bit of structure. It clearly poses no difficulty in the above procedures.

3.1.8 The Algebra of Triangular Matrices

For future reference we list a few properties about products and inverses of triangular and unit triangular matrices.

- The inverse of an upper (lower) triangular matrix is upper (lower) triangular.
- The product of two upper (lower) triangular matrices is upper (lower) triangular.
- The inverse of a unit upper (lower) triangular matrix is unit upper (lower) triangular.
- The product of two unit upper (lower) triangular matrices is unit upper (lower) triangular.

Problems

P3.1.1 Give an algorithm for computing a nonzero $z \in \mathbb{R}^n$ such that $Uz = 0$ where $U \in \mathbb{R}^{n\times n}$ is upper triangular with $u_{nn} = 0$ and $u_{11} \cdots u_{n-1,n-1} \neq 0$.

P3.1.2 Discuss how the determinant of a square triangular matrix could be computed with minimum risk of overflow and underflow.

P3.1.3 Rewrite Algorithm 3.1.4 given that U is stored by column in a length $n(n+1)/2$ array *u.vec*.

P3.1.4 Write a detailed version of (3.1.4). Do not assume that N divides n.

P3.1.5 Prove all the facts about triangular matrices that are listed in §3.1.8.

P3.1.6 Suppose $S, T \in \mathbb{R}^{n\times n}$ are upper triangular and that $(ST - \lambda I)x = b$ is a nonsingular system. Give an $O(n^2)$ algorithm for computing x. Note that the explicit formation of $ST - \lambda I$ requires $O(n^3)$ flops. Hint. Suppose

$$S_+ = \begin{bmatrix} \sigma & u^T \\ 0 & S_c \end{bmatrix}, \quad T_+ = \begin{bmatrix} \tau & v^T \\ 0 & T_c \end{bmatrix}, \quad b_+ = \begin{bmatrix} \beta \\ b_c \end{bmatrix}$$

where $S_+ = S(k-1{:}n, k-1{:}n)$, $T_+ = T(k-1{:}n, k-1{:}n)$, $b_+ = b(k-1{:}n)$, and $\sigma, \tau, \beta \in \mathbb{R}$. Show that if we have a vector x_c such that

$$(S_cT_c - \lambda I)x_c = b_c$$

and $w_c = T_cx_c$ is available, then

$$x_+ = \begin{bmatrix} \gamma \\ x_c \end{bmatrix} \qquad \gamma = \frac{\beta - \sigma v^T x_c - u^T w_c}{\sigma\tau - \lambda}$$

solves $(S_+T_+ - \lambda I)x_+ = b_+$. Observe that x_+ and $w_+ = T_+x_+$ each require $O(n-k)$ flops.

P3.1.7 Suppose the matrices $R_1, \ldots, R_p \in \mathbb{R}^{n\times n}$ are all upper triangular. Give an $O(pn^2)$ algorithm for solving the system $(R_1 \cdots R_p - \lambda I)x = b$ assuming that the matrix of coefficients is nonsingular. Hint. Generalize the solution to the previous problem.

Notes and References for Sec. 3.1

The accuracy of triangular system solvers is analyzed in

N.J. Higham (1989). "The Accuracy of Solutions to Triangular Systems," *SIAM J. Num. Anal.* 26, 1252–1265.

3.2 The LU Factorization

As we have just seen, triangular systems are "easy" to solve. The idea behind Gaussian elimination is to convert a given system $Ax = b$ to an equivalent triangular system. The conversion is achieved by taking appropriate linear combinations of the equations. For example, in the system

$$\begin{aligned} 3x_1 + 5x_2 &= 9 \\ 6x_1 + 7x_2 &= 4 \end{aligned}$$

if we multiply the first equation by 2 and subtract it from the second we obtain

$$\begin{aligned} 3x_1 + 5x_2 &= 9 \\ -3x_2 &= -14 \end{aligned}$$

This is $n = 2$ Gaussian elimination. Our objective in this section is to give a complete specification of this central procedure and to describe what it does in the language of matrix factorizations. This means showing that the algorithm computes a unit lower triangular matrix L and an upper triangular matrix U so that $A = LU$, e.g.,

$$\begin{bmatrix} 3 & 5 \\ 6 & 7 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}\begin{bmatrix} 3 & 5 \\ 0 & -3 \end{bmatrix}.$$

The solution to the original $Ax = b$ problem is then found by a two step triangular solve process:

$$Ly = b, \quad Ux = y \qquad \Longrightarrow \qquad Ax = LUx = Ly = b\,.$$

The LU factorization is a "high-level" algebraic description of Gaussian elimination. Expressing the outcome of a matrix algorithm in the "language" of matrix factorizations is a worthwhile activity. It facilitates generalization and highlights connections between algorithms that may appear very different at the scalar level.

3.2.1 Gauss Transformations

To obtain a factorization description of Gaussian elimination we need a matrix description of the zeroing process. At the $n = 2$ level if $x_1 \neq 0$ and $\tau = x_2/x_1$, then

$$\begin{bmatrix} 1 & 0 \\ -\tau & 1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_1 \\ 0 \end{bmatrix}$$

More generally, suppose $x \in \mathbb{R}^n$ with $x_k \neq 0$. If

$$\tau^T = (\underbrace{0,\ldots,0}_{k}, \tau_{k+1}, \ldots, \tau_n) \qquad \tau_i = \frac{x_i}{x_k} \quad i = k+1{:}n$$

and we define

$$M_k = I - \tau e_k^T, \tag{3.2.1}$$

then

$$M_k x = \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & & \vdots \\ 0 & & 1 & 0 & & 0 \\ 0 & & -\tau_{k+1} & 1 & & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & -\tau_n & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_k \\ x_{k+1} \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} x_1 \\ \vdots \\ x_k \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

In general, a matrix of the form $M_k = I - \tau e_k^T \in \mathbb{R}^{n \times n}$ is a *Gauss transformation* if the first k components of $\tau \in \mathbb{R}^n$ are zero. Such a matrix is unit lower triangular. The components of $\tau(k+1{:}n)$ are called *multipliers*. The vector τ is called the *Gauss vector*.

3.2.2 Applying Gauss Transformations

Multiplication by a Gauss transformation is particularly simple. If $C \in \mathbb{R}^{n \times r}$ and $M_k = I - \tau e_k^T$ is a Gauss transform, then

$$M_k C = (I - \tau e_k^T)C = C - \tau(e_k^T C) = C - \tau C(k,:).$$

is an outer product update. Since $\tau(1{:}k) = 0$ only $C(k+1{:}n,:)$ is affected and the update $C = M_k C$ can be computed row-by-row as follows:

```
for i = k+1:n
    C(i,:) = C(i,:) - τ_i C(k,:)
end
```

This computation requires $2(n-1)r$ flops.

Example 3.2.1

$$C = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 10 \end{bmatrix}, \tau = \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix} \Rightarrow (I - \tau e_1^T)C = \begin{bmatrix} 1 & 4 & 7 \\ 1 & 1 & 1 \\ 4 & 10 & 17 \end{bmatrix}.$$

3.2.3 Roundoff Properties of Gauss Transforms

If $\hat{\tau}$ is the computed version of an exact Gauss vector τ, then it is easy to verify that

$$\hat{\tau} = \tau + e \qquad |e| \le \mathbf{u}|\tau|.$$

If $\hat{\tau}$ is used in a Gauss transform update and $fl((I - \hat{\tau}e_k^T)C)$ denotes the computed result, then

$$fl\left((I - \hat{\tau}e_k^T)C\right) = (I - \tau e_k^T)C + E,$$

where

$$|E| \le 3\mathbf{u}\left(|C| + |\tau||C(k,:)|\right) + O(\mathbf{u}^2).$$

Clearly, if τ has large components, then the errors in the update may be large in comparison to $|C|$. For this reason, care must be exercised when Gauss transformations are employed, a matter that is pursued in §3.4.

3.2.4 Upper Triangularizing

Assume that $A \in \mathbb{R}^{n \times n}$. Gauss transformations $M_1, \ldots, M_{n-1}$ can usually be found such that $M_{n-1} \cdots M_2 M_1 A = U$ is upper triangular. To see this we first look at the $n = 3$ case. Suppose

$$A = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 10 \end{bmatrix}.$$

If

$$M_1 = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -3 & 0 & 1 \end{bmatrix},$$

then

$$M_1 A = \begin{bmatrix} 1 & 4 & 7 \\ 0 & -3 & -6 \\ 0 & -6 & -11 \end{bmatrix}.$$

Likewise,

$$M_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2 & 1 \end{bmatrix} \quad \Rightarrow \quad M_2(M_1 A) = \begin{bmatrix} 1 & 4 & 7 \\ 0 & -3 & -6 \\ 0 & 0 & 1 \end{bmatrix}.$$

Extrapolating from this example observe that during the kth step

- We are confronted with a matrix $A^{(k-1)} = M_{k-1} \cdots M_1 A$ that is upper triangular in columns 1 to $k-1$.
- The multipliers in M_k are based on $A^{(k-1)}(k+1{:}n, k)$. In particular, we need $a_{kk}^{(k-1)} \neq 0$ to proceed.

Noting that complete upper triangularization is achieved after $n-1$ steps we therefore obtain

```
k = 1
while (A(k,k) ≠ 0) & (k ≤ n − 1)
      τ(k + 1:n) = A(k + 1:n, k)/A(k,k)                        (3.2.2)
      A(k + 1:n, :) = A(k + 1:n, :) − τ(k + 1:n)A(k, :)
      k = k + 1
end
```

The entry $A(k,k)$ must be checked to avoid a zero divide. These quantities are referred to as the *pivots* and their relative magnitude turns out to be critically important.

3.2.5 The LU Factorization

In matrix language, if (3.2.2) terminates with $k = n$, then it computes Gauss transforms $M_1, \ldots, M_{n-1}$ such that $M_{n-1} \cdots M_1 A = U$ is upper triangular. It is easy to check that if $M_k = I - \tau^{(k)} e_k^T$, then its inverse is prescribed by $M_k^{-1} = I + \tau^{(k)} e_k^T$ and so

$$A = LU \tag{3.2.3}$$

where

$$L = M_1^{-1} \cdots M_{n-1}^{-1}. \tag{3.2.4}$$

It is clear that L is a unit lower triangular matrix because each M_k^{-1} is unit lower triangular. The factorization (3.2.3) is called the *LU factorization* of A.

As suggested by the need to check for zero pivots in (3.2.2), the LU factorization need not exist. For example, it is impossible to find l_{ij} and u_{ij} so

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 7 \\ 3 & 5 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \ell_{21} & 1 & 0 \\ \ell_{31} & \ell_{32} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}.$$

To see this equate entries and observe that we must have $u_{11} = 1$, $u_{12} = 2$, $\ell_{21} = 2$, $u_{22} = 0$, and $\ell_{31} = 3$. But when we then look at the (3,2) entry we obtain the contradictory equation $5 = \ell_{31}u_{12} + \ell_{32}u_{22} = 6$.

As we now show, a zero pivot in (3.2.2) can be identified with a singular leading principal submatrix.

Theorem 3.2.1 *$A \in \mathbb{R}^{n\times n}$ has an LU factorization if* $\det(A(1{:}k, 1{:}k)) \neq 0$ *for $k = 1{:}n-1$. If the LU factorization exists and A is nonsingular, then the LU factorization is unique and* $\det(A) = u_{11} \cdots u_{nn}$.

Proof. Suppose $k-1$ steps in (3.2.2) have been executed. At the beginning of step k the matrix A has been overwritten by $M_{k-1} \cdots M_1 A = A^{(k-1)}$. Note that $a_{kk}^{(k-1)}$ is the kth pivot. Since the Gauss transformations are unit lower triangular it follows by looking at the leading k-by-k portion of this equation that $\det(A(1{:}k, 1{:}k)) = a_{11}^{(k-1)} \cdots a_{kk}^{(k-1)}$. Thus, if $A(1{:}k, 1{:}k)$ is nonsingular then the kth pivot is nonzero.

As for uniqueness, if $A = L_1U_1$ and $A = L_2U_2$ are two LU factorizations of a nonsingular A, then $L_2^{-1}L_1 = U_2U_1^{-1}$. Since $L_2^{-1}L_1$ is unit lower triangular and $U_2U_1^{-1}$ is upper triangular, it follows that both of these matrices must equal the identity. Hence, $L_1 = L_2$ and $U_1 = U_2$.

Finally, if $A = LU$ then $\det(A) = \det(LU) = \det(L)\det(U) = \det(U) = u_{11} \cdots u_{nn}$. □

3.2.6 Some Practical Details

From the practical point of view there are several improvements that can be made to (3.2.2). First, because zeros have already been introduced in columns 1 through $k-1$, the Gauss transform update need only be applied to columns k through n. Of course, we need not even apply the kth Gauss transform to $A(:,k)$ since we know the result. So the efficient thing to do is simply to update $A(k+1{:}n, k+1{:}n)$. Another worthwhile observation is that the multipliers associated with M_k can be stored in the locations that they zero, i.e., $A(k+1{:}n, k)$. With these changes we obtain the following version of (3.2.2):

Algorithm 3.2.1 (Outer Product Gaussian Elimination) Suppose $A \in \mathbb{R}^{n\times n}$ has the property that $A(1{:}k, 1{:}k)$ is nonsingular for $k = 1{:}n-1$. This algorithm computes the factorization $M_{n-1} \cdots M_1 A = U$ where U is upper triangular and each M_k is a Gauss transform. U is stored in the upper triangle of A. The multipliers associated with M_k are stored in $A(k+1{:}n, k)$, i.e., $A(k+1{:}n, k) = -M_k(k+1{:}n, k)$.

```
for k = 1:n − 1
      rows = k + 1:n
      A(rows, k) = A(rows, k)/A(k, k)
      A(rows, rows) = A(rows, rows) − A(rows, k)A(k, rows)
end
```

This algorithm involves $2n^3/3$ flops and it is one of several formulations of *Gaussian Elimination*. Note that each pass through the k-loop involves an outer product.

3.2.7 Where is L?

Algorithm 3.2.3 represents L in terms of the multipliers. In particular, if $\tau^{(k)}$ is the vector of multipliers associated with M_k then upon termination, $A(k+1{:}n, k) = \tau^{(k)}$. One of the more happy "coincidences" in matrix

computations is that if $L = M_1^{-1}\cdots M_{n-1}^{-1}$, then $L(k+1{:}n,k) = \tau^{(k)}$. This follows from a careful look at the product that defines L. Indeed,

$$L = \left(I + \tau^{(1)}e_1^T\right)\cdots\left(I + \tau^{(n-1)}e_{n-1}^T\right) \;=\; I + \sum_{k=1}^{n-1}\tau^{(k)}e_k^T\,.$$

Since $A(k+1{:}n,k)$ houses the kth vector of multipliers $\tau^{(k)}$, it follows that $A(i,k)$ houses ℓ_{ik} for all $i > k$.

3.2.8 Solving a Linear System

Once A has been factored via Algorithm 3.2.1, then L and U are represented in the array A. We can then solve the system $Ax = b$ via the triangular systems $Ly = b$ and $Ux = y$ by using the methods of §3.1.

Example 3.2.2 If Algorithm 3.2.1 is applied to

$$A = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 10 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 2 & 1 \end{bmatrix}\begin{bmatrix} 1 & 4 & 7 \\ 0 & -3 & -6 \\ 0 & 0 & 1 \end{bmatrix},$$

then upon completion,

$$A = \begin{bmatrix} 1 & 4 & 7 \\ 2 & -3 & -6 \\ 3 & 2 & 1 \end{bmatrix}.$$

If $b = (1,1,1)^T$, then $y = (1,-1,0)^T$ solves $Ly = b$ and $x = (-1/3, 1/3, 0)^T$ solves $Ux = y$.

3.2.9 Other Versions

Gaussian elimination, like matrix multiplication, is a triple-loop procedure that can be arranged in several ways. Algorithm 3.2.1 corresponds to the "kij" version of Gaussian elimination if we compute the outer product update row-by-row:

```
for k = 1:n-1
    A(k+1:n,k) = A(k+1:n,k)/A(k,k)
    for i = k+1:n
        for j = k+1:n
            A(i,j) = A(i,j) - A(i,k)A(k,j)
        end
    end
end
```

There are five other versions: kji, ikj, ijk, jik, and jki. The last of these results in an implementation that features a sequence of gaxpy's and forward eliminations. In this formulation, the Gauss transformations are not immediately applied to A as they are in the outer product version. Instead, their application is delayed. The original $A(:,j)$ is untouched until step j. At that point in the algorithm $A(:,j)$ is overwritten by $M_{j-1}\cdots M_1A(:,j)$. The jth Gauss transformation is then computed.

To be precise, suppose $1 \le j \le n-1$ and assume that $L(:,1{:}j-1)$ and $U(1{:}j-1,1{:}j-1)$ are known. This means that the first $j-1$ columns of L and U are available. To get the jth columns of L and U we equate jth columns in the equation $A = LU$: $A(:,j) = LU(:,j)$. From this we conclude that

$$A(1{:}j-1,j) \;=\; L(1{:}j-1,1{:}j-1)U(1{:}j-1,j)$$

and

$$A(j{:}n,j) \;=\; \sum_{k=1}^{j} L(j{:}n,k)U(k,j)\,.$$

The first equation is a lower triangular system that can be solved for the vector $U(1{:}j-1,j)$. Once this is accomplished, the second equation can be rearranged to produce recipes for $U(j,j)$ and $L(j+1{:}n,j)$. Indeed, if we set

$$\begin{aligned} v(j{:}n) \;&=\; A(j{:}n,j) - \sum_{k=1}^{j-1} L(j{:}n,k)U(k,j) \\ &=\; A(j{:}n,j) - L(j{:}n,1{:}j-1)U(1{:}j-1,j), \end{aligned}$$

then $L(j+1{:}n,j) = v(j+1{:}n)/v(j)$ and $U(j,j) = v(j)$. Thus, $L(j+1{:}n,j)$ is a scaled gaxpy and we obtain

```
L = I; U = 0
for j = 1:n
    if j = 1
        v(j:n) = A(j:n,j)
    else
        Solve L(1:j-1,1:j-1)z = A(1:j-1,j) for z          (3.2.5)
            and set U(1:j-1,j) = z.
        v(j:n) = A(j:n,j) - L(j:n,1:j-1)z
    end
    if j < n
        L(j+1:n,j) = v(j+1:n)/v(j)
    end
    U(j,j) = v(j)
end
```

This arrangement of Gaussian elimination is rich in forward eliminations and gaxpy operations and, like Algorithm 3.2.1, requires $2n^3/3$ flops.

3.2.10 Block LU

It is possible to organize Gaussian elimination so that matrix multiplication becomes the dominant operation. The key to the derivation of this block procedure is to partition $A \in \mathbb{R}^{n \times n}$ as follows

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{matrix} r \\ n-r \end{matrix}$$
$$\qquad\quad r \quad n-r$$

where r is a blocking parameter. Suppose we compute the LU factorization $L_{11}U_{11} = A_{11}$ and then solve the multiple right hand side triangular systems $L_{11}U_{12} = A_{12}$ and $L_{21}U_{11} = A_{21}$ for U_{12} and L_{21} respectively. It follows that

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I_{n-r} \end{bmatrix} \begin{bmatrix} I_r & 0 \\ 0 & \tilde{A} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & I_{n-r} \end{bmatrix}$$

where $\tilde{A} = A_{22} - L_{21}U_{12}$. The matrix $\tilde{A}$ is the *Schur complement* of A_{11} with respect to A. Note that if $\tilde{A} = L_{22}U_{22}$ is the *LU* factorization of $\tilde{A}$, then

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} I_r & 0 \\ 0 & \tilde{A} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix}$$

is the *LU* factorization of A. Thus, after L_{11}, L_{21}, U_{11} and U_{22}, are computed, we repeat the process on the level-3 updated (2,2) block $\tilde{A}$.

Algorithm 3.2.2 (Block Outer Product LU) Suppose $A \in \mathbb{R}^{n \times n}$ and that $\det(A(1{:}k, 1{:}k))$ is nonzero for $k = 1{:}n-1$. Assume that r satisfies $1 \le r \le n$. The following algorithm computes $A = LU$ via rank r updates. Upon completion, $A(i,j)$ is overwritten with $L(i,j)$ for $i > j$ and $A(i,j)$ is overwritten with $U(i,j)$ if $j \ge i$.

```
λ = 1
while λ ≤ n
      μ = min(n, λ + r − 1)
      Use Algorithm 3.2.1 to overwrite A(λ:μ, λ:μ)
            with its LU factors L̃ and Ũ.
      Solve L̃Z = A(λ:μ, μ + 1:n) for Z and overwrite
            A(λ:μ, μ + 1:n) with Z.
      Solve WŨ = A(μ + 1:n, λ:μ) for W and overwrite
            A(μ + 1:n, λ:μ) with W.
      A(μ + 1:n, μ + 1:n) = A(μ + 1:n, μ + 1:n) − WZ
      λ = μ + 1
end
```


This algorithm involves $2n^3/3$ flops.

Recalling the discussion in §3.1.5, let us consider the level-3 fraction for this procedure assuming that r is large enough so that the underlying computer is able to compute the matrix multiply update $A(\mu+1{:}n, \mu+1{:}n) = A(\mu+1{:}n, \mu+1{:}n) - WZ$ at "level-3 speed." Assume for clarity that $n = rN$. The only flops that are not level-3 flops occur in the context of the r-by-r LU factorizations $A(\lambda{:}\mu, \lambda{:}\mu) = \tilde{L}\tilde{U}$. Since there are N such systems solved in the overall computation, we see that the level-3 fraction is given by

$$1 - \frac{N(2r^3/3)}{2n^3/3} = 1 - \frac{1}{N^2}.$$

Thus, for large N almost all arithmetic takes place in the context of matrix multiplication. As we have mentioned, this ensures high performance on a wide range of computing environments.

3.2.11 The LU Factorization of a Rectangular Matrix

The LU factorization of a rectangular matrix $A \in \mathbb{R}^{m \times n}$ can also be performed. The $m > n$ case is illustrated by

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 3 & 1 \\ 5 & 2 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 0 & -2 \end{bmatrix}$$

while

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \end{bmatrix}$$

depicts the $m < n$ situation. The LU factorization of $A \in \mathbb{R}^{m \times n}$ is guaranteed to exist if $A(1{:}k, 1{:}k)$ is nonsingular for $k = 1{:}\min(m, n)$.

The square LU factorization algorithms above need only minor modification to handle the rectangular case. For example, to handle the $m > n$ case we modify Algorithm 3.2.1 as follows:

```
for k = 1:n
      rows = k + 1:m
      A(rows, k) = A(rows, k)/A(k, k)
      if k < n
            cols = k + 1:n
            A(rows, cols) = A(rows, cols) − A(rows, k)A(k, cols)
      end
end
```

This algorithm requires $mn^2 - n^3/3$ flops.

3.2.12 A Note on Failure

As we know, Gaussian elimination fails unless the first $n-1$ principal submatrices are nonsingular. This rules out some very simple matrices, e.g.,

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

While A has perfect 2-norm condition, it fails to have an LU factorization because it has a singular leading principal submatrix.

Clearly, modifications are necessary if Gaussian elimination is to be effectively used in general linear system solving. The error analysis in the following section suggests the needed modifications.

Problems

P3.2.1 Suppose the entries of $A(\epsilon) \in \mathbb{R}^{n\times n}$ are continuously differentiable functions of the scalar ϵ. Assume that $A \equiv A(0)$ and all its principal submatrices are nonsingular. Show that for sufficiently small ϵ, the matrix $A(\epsilon)$ has an LU factorization $A(\epsilon) = L(\epsilon)U(\epsilon)$ and that $L(\epsilon)$ and $U(\epsilon)$ are both continuously differentiable.

P3.2.2 Suppose we partition $A \in \mathbb{R}^{n\times n}$

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

where A_{11} is r-by-r. Assume that A_{11} is nonsingular. The matrix $S = A_{22} - A_{21}A_{11}^{-1}A_{12}$ is called the *Schur complement* of A_{11} in A. Show that if A_{11} has an LU factorization, then after r steps of Algorithm 3.2.1, $A(r+1{:}n, r+1{:}n)$ houses S. How could S be obtained after r steps of (3.2.5)?

P3.2.3 Suppose $A \in \mathbb{R}^{n\times n}$ has an LU factorization. Show how $Ax = b$ can be solved without storing the multipliers by computing the LU factorization of the n-by-$(n+1)$ matrix $[A\ b]$.

P3.2.4 Describe a variant of Gaussian elimination that introduces zeros into the columns of A in the order, $n{:}-1{:}2$ and which produces the factorization $A = UL$ where U is unit upper triangular and L is lower triangular.

P3.2.5 Matrices in $\mathbb{R}^{n\times n}$ of the form $N(y,k) = I - ye_k^T$ where $y \in \mathbb{R}^n$ are said to be *Gauss-Jordan transformations*. (a) Give a formula for $N(y,k)^{-1}$ assuming it exists. (b) Given $x \in \mathbb{R}^n$, under what conditions can y be found so $N(y,k)x = e_k$? (c) Give an algorithm using Gauss-Jordan transformations that overwrites A with A^{-1}. What conditions on A ensure the success of your algorithm?

P3.2.6 Extend (3.2.5) so that it can also handle the case when A has more rows than columns.

P3.2.7 Show how A can be overwritten with L and U in (3.2.5). Organize the three loops so that unit stride access prevails.

P3.2.8 Develop a version of Gaussian elimination in which the innermost of the three loops oversees a dot product.

Notes and References for Sec. 3.2

Schur complements (P3.2.2) arise in many applications. For a survey of both practical and theoretical interest, see

R.W. Cottle (1974). "Manifestations of the Schur Complement," *Lin. Alg. and Its Applic.* 8, 189-211.

Schur complements are known as "Gauss transforms" in some application areas. The use of Gauss-Jordan transformations (P3.2.5) is detailed in Fox (1964). See also

T. Dekker and W. Hoffman (1989). "Rehabilitation of the Gauss-Jordan Algorithm," *Numer. Math.* 54, 591–599.

As we mentioned, inner product versions of Gaussian elimination have been known and used for some time. The names of Crout and Doolittle are associated with these ijk techniques. They were popular during the days of desk calculators because there are far fewer intermediate results than in Gaussian elimination. These methods still have attraction because they can be implemented with accumulated inner products. For remarks along these lines see Fox (1964) as well as Stewart (1973, pp. 131–39). See also:

G.E. Forsythe (1960). "Crout with Pivoting," *Comm. ACM* 3, 507–8.
W.M. McKeeman (1962). "Crout with Equilibration and Iteration," *Comm. ACM.* 5, 553–55.

Loop orderings and block issues in LU computations are discussed in

J.J. Dongarra, F.G. Gustavson, and A. Karp (1984). "Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine," *SIAM Review* 26, 91–112.
J.M. Ortega (1988). "The ijk Forms of Factorization Methods I: Vector Computers," *Parallel Computers* 7, 135–147.
D.H. Bailey, K.Lee, and H.D. Simon (1991). "Using Strassen's Algorithm to Accelerate the Solution of Linear Systems," *J. Supercomputing* 4, 357–371.
J.W. Demmel, N.J. Higham, and R.S. Schreiber (1995). "Stability of Block LU Factorization," *Numer. Lin. Alg. with Applic.* 2, 173–190.

3.3 Roundoff Analysis of Gaussian Elimination

We now assess the effect of rounding errors when the algorithms in the previous two sections are used to solve the linear system $Ax = b$. A much more detailed treatment of roundoff error in Gaussian elimination is given in Higham (1996).

Before we proceed with the analysis, it is useful to consider the nearly ideal situation in which no roundoff occurs during the entire solution process except when A and b are stored. Thus, if $fl(b) = b+e$ and the stored matrix $fl(A) = A + E$ is nonsingular, then we are assuming that the computed solution $\hat{x}$ satisfies

$$(A+E)\hat{x} = (b+e) \qquad \| E \|_\infty \le \mathbf{u}\| A \|_\infty, \quad \| e \|_\infty \le \mathbf{u}\| b \|_\infty . \quad (3.3.1)$$

That is, $\hat{x}$ solves a "nearby" system exactly. Moreover, if $\mathbf{u}\kappa_\infty(A) \le \frac{1}{2}$ (say), then by using Theorem 2.7.2, it can be shown that

$$\frac{\| x - \hat{x} \|_\infty}{\| x \|_\infty} \le 4\mathbf{u}\kappa_\infty(A) . \tag{3.3.2}$$

The bounds (3.3.1) and (3.3.2) are "best possible" norm bounds. No general ∞-norm error analysis of a linear equation solver that requires the storage of A and b can render sharper bounds. As a consequence, we cannot justifiably criticize an algorithm for returning an inaccurate $\hat{x}$ if A is ill-conditioned relative to the machine precision, e.g., $\mathbf{u}\kappa_\infty(A) \approx 1$.

3.3.1 Errors in the LU Factorization

Let us see how the error bounds for Gaussian elimination compare with the ideal bounds above. We work with the infinity norm for convenience and focus our attention on Algorithm 3.2.3, the outer product version. The error bounds that we derive also apply to Algorithm 3.2.4, the gaxpy formulation.

Our first task is to quantify the roundoff errors associated with the computed triangular factors.

Theorem 3.3.1 *Assume that A is an n-by-n matrix of floating point numbers. If no zero pivots are encountered during the execution of Algorithm 3.2.3, then the computed triangular matrices $\hat{L}$ and $\hat{U}$ satisfy*

$$\hat{L}\hat{U} = A + H \tag{3.3.3}$$

$$|H| \le 3(n-1)\mathbf{u}\left(|A| + |\hat{L}||\hat{U}|\right) + O(\mathbf{u}^2) . \tag{3.3.4}$$

Proof. The proof is by induction on n. The theorem obviously holds for $n = 1$. Assume it holds for all $(n-1)$-by-$(n-1)$ floating point matrices. If

$$A = \begin{bmatrix} \alpha & w^T \\ v & B \end{bmatrix} \begin{matrix} 1 \\ n-1 \end{matrix}$$
$$\quad\;\; 1 \quad n-1$$

then $\hat{z} = fl(v/\alpha)$ and $\hat{A}_1 = fl(B - \hat{z}w^T)$ are computed in the first step of the algorithm. We therefore have

$$\hat{z} = \frac{1}{\alpha}v + f \qquad |f| \le \mathbf{u}\frac{|v|}{|\alpha|} \tag{3.3.5}$$

and

$$\hat{A}_1 = B - \hat{z}w^T + F \qquad |F| \le 2\mathbf{u}\left(|B| + |\hat{z}||w|^T\right) + O(\mathbf{u}^2) . \tag{3.3.6}$$

The algorithm now proceeds to calculate the LU factorization of $\hat{A}_1$. By induction, we compute approximate factors $\hat{L}_1$ and $\hat{U}_1$ for $\hat{A}_1$ that satisfy

$$\hat{L}_1\hat{U}_1 = \hat{A}_1 + H_1 \tag{3.3.7}$$

$$|H_1| \le 3(n-2)\mathbf{u}\left(|\hat{A}_1| + |\hat{L}_1||\hat{U}_1|\right) + O(\mathbf{u}^2). \tag{3.3.8}$$

Thus,

$$\begin{aligned} \hat{L}\hat{U} &\equiv \begin{bmatrix} 1 & 0 \\ \hat{z} & \hat{L}_1 \end{bmatrix} \begin{bmatrix} \alpha & w^T \\ 0 & \hat{U}_1 \end{bmatrix} \\ &= A + \begin{bmatrix} 0 & 0 \\ \alpha f & H_1 + F \end{bmatrix} \equiv A + H . \end{aligned}$$

From (3.3.6) it follows that

$$|\hat{A}_1| \le (1 + 2\mathbf{u})\left(|B| + |\hat{z}||w|^T\right) + O(\mathbf{u}^2),$$

and therefore by using (3.3.7) and (3.3.8) we have

$$|H_1 + F| \le 3(n-1)\mathbf{u}\left(|B| + |\hat{z}||w|^T + |\hat{L}_1||\hat{U}_1|\right) + O(\mathbf{u}^2).$$

Since $|\alpha f| \le \mathbf{u}|v|$ it is easy to verify that

$$|H| \le 3(n-1)\mathbf{u}\left\{\begin{bmatrix} |\alpha| & |w|^T \\ |v| & |B| \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ |\hat{z}| & |\hat{L}_1| \end{bmatrix}\begin{bmatrix} |\alpha| & |w|^T \\ 0 & |\hat{U}_1| \end{bmatrix}\right\} + O(\mathbf{u}^2)$$

thereby proving the theorem. □

We mention that if A is m-by-n, then the theorem applies with n in (3.3.4) replaced by the smaller of n and m .

3.3.2 Triangular Solving with Inexact Triangles

We next examine the effect of roundoff error when $\hat{L}$ and $\hat{U}$ are used by the triangular system solvers of §3.1.

Theorem 3.3.2 *Let $\hat{L}$ and $\hat{U}$ be the computed LU factors of the n-by-n floating point matrix A obtained by either Algorithm 3.2.3 or 3.2.4. Suppose the methods of §3.1 are used to produce the computed solution $\hat{y}$ to $\hat{L}y = b$ and the computed solution $\hat{x}$ to $\hat{U}x = \hat{y}$. Then $(A + E)\hat{x} = b$ with*

$$|E| \le n\mathbf{u}\left(3|A| + 5|\hat{L}||\hat{U}|\right) + O(\mathbf{u}^2) . \tag{3.3.9}$$

Proof. From (3.1.1) and (3.1.2) we have

$$\begin{aligned} (\hat{L}+F)\hat{y} &= b & \quad |F| &\le n\mathbf{u}|\hat{L}| + O(\mathbf{u}^2) \\ (\hat{U}+G)\hat{x} &= \hat{y} & \quad |G| &\le n\mathbf{u}|\hat{U}| + O(\mathbf{u}^2) \end{aligned}$$

and thus

$$(\hat{L}+F)(\hat{U}+G)\hat{x} \;=\; (\hat{L}\hat{U} + F\hat{U} + \hat{L}G + FG)\hat{x} = b.$$

From Theorem 3.3.1

$$\hat{L}\hat{U} = A + H,$$

with $|H| \le 3(n-1)\mathbf{u}(|A| + |\hat{L}||\hat{U}|) + O(\mathbf{u}^2)$, and so by defining

$$E \;=\; H + F\hat{U} + \hat{L}G + FG$$

we find $(A+E)\hat{x} = b$. Moreover,

$$\begin{aligned} |E| &\le |H| + |F|\,|\hat{U}| \;+\; |\hat{L}|\,|G| \;+\; O(\mathbf{u}^2) \\ &\le 3n\mathbf{u}\left(|A| + |\hat{L}||\hat{U}|\right) + 2n\mathbf{u}\left(|\hat{L}||\hat{U}|\right) + O(\mathbf{u}^2). \quad \square \end{aligned}$$

Were it not for the possibility of a large $|\hat{L}||\hat{U}|$ term, (3.3.9) would compare favorably with the ideal bound in (3.3.1). (The factor n is of no consequence, cf. the Wilkinson quotation in §2.4.6.) Such a possibility exists, for there is nothing in Gaussian elimination to rule out the appearance of small pivots. If a small pivot is encountered, then we can expect large numbers to be present in $\hat{L}$ and $\hat{U}$.

We stress that small pivots are not necessarily due to ill-conditioning as the example $A = \begin{bmatrix} \epsilon & 1 \\ 1 & 0 \end{bmatrix}$ bears out. Thus, Gaussian elimination can give arbitrarily poor results, even for well-conditioned problems. The method is unstable.

In order to repair this shortcoming of the algorithm, it is necessary to introduce row and/or column interchanges during the elimination process with the intention of keeping the numbers that arise during the calculation suitably bounded. This idea is pursued in the next section.

Example 3.3.1 Suppose $\beta = 10, t = 3$, floating point arithmetic is used to solve:

$$\begin{bmatrix} .001 & 1.00 \\ 1.00 & 2.00 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1.00 \\ 3.00 \end{bmatrix}.$$

Applying Gaussian elimination we get

$$\hat{L} = \begin{bmatrix} 1 & 0 \\ 1000 & 1 \end{bmatrix} \qquad \hat{U} = \begin{bmatrix} .001 & 1 \\ 0 & -1000 \end{bmatrix}$$

and a calculation shows

$$\hat{L}\hat{U} = \begin{bmatrix} .001 & 1 \\ 1 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & -2 \end{bmatrix} \equiv A + H.$$

Moreover, $6\begin{bmatrix} 10^{-6} & .001 \\ 10^{-3} & 1.0001 \end{bmatrix}$ is the bounding matrix in (3.3.4), not a severe overestimate of $|H|$. If we go on to solve the problem using the triangular system solvers of §3.1, then using the same precision arithmetic we obtain a computed solution $\hat{x} = (0,1)^T$. This is in contrast to the exact solution $x = (1.002\ldots, .998\ldots)^T$.

Problems

P3.3.1 Show that if we drop the assumption that A is a floating point matrix in Theorem 3.3.1, then (3.3.4) holds with the coefficient "3" replaced by "4."

P3.3.2 Suppose A is an n-by-n matrix and that $\hat{L}$ and $\hat{U}$ are produced by Algorithm 3.2.1. (a) How many flops are required to compute $\|\,|\hat{L}|\,|\hat{U}|\,\|_\infty$? (b) Show $fl(|\hat{L}||\hat{U}|) \le (1+2n\mathbf{u})|\hat{L}||\hat{U}| + O(\mathbf{u}^2)$.

P3.3.3 Suppose $x = A^{-1}b$. Show that if $e = x - \hat{x}$ (the error) and $r = b - A\hat{x}$ (the residual), then

$$\frac{\|r\|}{\|A\|} \le \|e\| \le \|A^{-1}\|\,\|r\|.$$

Assume consistency between the matrix and vector norm.

P3.3.4 Using 2-digit, base 10, floating point arithmetic, compute the LU factorization of

$$A = \begin{bmatrix} 7 & 6 \\ 9 & 8 \end{bmatrix}.$$

For this example, what is the matrix H in (3.3.3)?

Notes and References for Sec. 3.3

The original roundoff analysis of Gaussian elimination appears in

J.H. Wilkinson (1961). "Error Analysis of Direct Methods of Matrix Inversion," *J. ACM 8*, 281-330.

Various improvements in the bounds and simplifications in the analysis have occurred over the years. See

B.A. Chartres and J.C. Geuder (1967). "Computable Error Bounds for Direct Solution of Linear Equations," *J. ACM 14*, 63–71.

J.K. Reid (1971). "A Note on the Stability of Gaussian Elimination," *J. Inst. Math. Applic. 8*, 374–75.

C.C. Paige (1973). "An Error Analysis of a Method for Solving Matrix Equations," *Math. Comp. 27*, 355–59.

C. de Boor and A. Pinkus (1977). "A Backward Error Analysis for Totally Positive Linear Systems," *Numer. Math. 27*, 485-90.

H.H. Robertson (1977). "The Accuracy of Error Estimates for Systems of Linear Algebraic Equations," *J. Inst. Math. Applic. 20*, 409–14.

J.J. Du Croz and N.J. Higham (1992). "Stability of Methods for Matrix Inversion," *IMA J. Num. Anal. 12*, 1–19.

3.4 Pivoting

The analysis in the previous section shows that we must take steps to ensure that no large entries appear in the computed triangular factors $\hat{L}$ and $\hat{U}$. The example

$$A = \begin{bmatrix} .0001 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 10,000 & 1 \end{bmatrix} \begin{bmatrix} .0001 & 1 \\ 0 & -9999 \end{bmatrix} = LU$$

correctly identifies the source of the difficulty: relatively small pivots. A way out of this difficulty is to interchange rows. In our example, if P is the permutation

$$P = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

then

$$PA = \begin{bmatrix} 1 & 1 \\ .0001 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ .0001 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & .9999 \end{bmatrix} = LU.$$

Now the triangular factors are comprised of acceptably small elements.

In this section we show how to determine a permuted version of A that has a reasonably stable LU factorization. There are several ways to do this and they each correspond to a different pivoting strategy. We focus on partial pivoting and complete pivoting. The efficient implementation of these strategies and their properties are discussed. We begin with a discussion of permutation matrix manipulation.

3.4.1 Permutation Matrices

The stabilizations of Gaussian elimination that are developed in this section involve data movements such as the interchange of two matrix rows. In keeping with our desire to describe all computations in "matrix terms," it is necessary to acquire a familiarity with *permutation matrices*. A permutation matrix is just the identity with its rows re-ordered, e.g.,

$$P = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}.$$

An n-by-n permutation matrix should never be explicitly stored. It is much more efficient to represent a general permutation matrix P with an integer n-vector p. One way to do this is to let $p(k)$ be the column index of the sole "1" in P's kth row. Thus, $p = [4\ 1\ 3\ 2]$ is the appropriate encoding of the above P. It is also possible to encode P on the basis of where the "1" occurs in each column, e.g., $p = [2\ 4\ 3\ 1]$.

If P is a permutation and A is a matrix, then PA is a row permuted version of A and AP is a column permuted version of A. Permutation matrices are orthogonal and so if P is a permutation, then $P^{-1} = P^T$. A product of permutation matrices is a permutation matrix.

In this section we are particularly interested in *interchange permutations*. These are permutations obtained by merely swapping two rows in the identity, e.g.,

$$E = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

Interchange permutations can be used to describe row and column swapping. With the above 4-by-4 example, EA is A with rows 1 and 4 interchanged. Likewise, AE is A with columns 1 and 4 swapped.

If $P = E_n \cdots E_1$ and each E_k is the identity with rows k and $p(k)$ interchanged, then $p(1{:}n)$ is a useful vector encoding of P. Indeed, $x \in \mathbb{R}^n$ can be overwritten by Px as follows:

```
for k = 1:n
    x(k) ↔ x(p(k))
end
```

Here, the "$\leftrightarrow$" notation means "swap contents." Since each E_k is symmetric and $P^T = E_1 \cdots E_n$, the representation can also be used to overwrite x with $P^T x$:

```
for k = n: − 1:1
    x(k) ↔ x(p(k))
end
```

It should be noted that no floating point arithmetic is involved in a permutation operation. However, permutation matrix operations often involve the irregular movement of data and can represent a significant computational overhead.

3.4.2 Partial Pivoting: The Basic Idea

We show how interchange permutations can be used in LU computations to guarantee that no multiplier is greater than one in absolute value. Suppose

$$A = \begin{bmatrix} 3 & 17 & 10 \\ 2 & 4 & -2 \\ 6 & 18 & -12 \end{bmatrix}.$$

To get the smallest possible multipliers in the first Gauss transform using row interchanges we need a_{11} to be the largest entry in the first column. Thus, if E_1 is the interchange permutation

$$E_1 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

then

$$E_1A = \begin{bmatrix} 6 & 18 & -12 \\ 2 & 4 & -2 \\ 3 & 17 & 10 \end{bmatrix}$$

and

$$M_1 = \begin{bmatrix} 1 & 0 & 0 \\ -1/3 & 1 & 0 \\ -1/2 & 0 & 1 \end{bmatrix} \implies M_1E_1A = \begin{bmatrix} 6 & 18 & -12 \\ 0 & -2 & 2 \\ 0 & 8 & 16 \end{bmatrix}.$$

Now to get the smallest possible multiplier in M_2 we need to swap rows 2 and 3. Thus, if

$$E_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \text{and} \quad M_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1/4 & 1 \end{bmatrix}$$

then

$$M_2E_2M_1E_1A = \begin{bmatrix} 6 & 18 & -12 \\ 0 & 8 & 16 \\ 0 & 0 & 6 \end{bmatrix}.$$

The example illustrates the basic idea behind the row interchanges. In general we have:

for $k = 1{:}n-1$
  Determine an interchange matrix E_k with $E_k(1{:}k, 1{:}k) = I_k$
    such that if z is the kth column of E_kA, then
    $|z(k)| = \| z(k{:}n) \|_\infty$.
  $A = E_kA$
  Determine the Gauss transform M_k such that if v is the
    kth column of M_kA, then $v(k+1{:}n) = 0$.
  $A = M_kA$
end

This particular row interchange strategy is called *partial pivoting*. Upon completion we emerge with $M_{n-1}E_{n-1}\cdots M_1E_1A = U$, an upper triangular matrix.

As a consequence of the partial pivoting, no multiplier is larger than one in absolute value. This is because

$$|(E_kM_{k-1}\cdots M_1E_1A)_{kk}| = \max_{k\le i\le n} |(E_kM_{k-1}\cdots M_1E_1A)_{ik}|$$

for $k = 1{:}n-1$. Thus, partial pivoting effectively guards against arbitrarily large multipliers.

3.4.3 Partial Pivoting Details

We are now set to detail the overall Gaussian Elimination with partial pivoting algorithm.

Algorithm 3.4.1 (Gauss Elimination with Partial Pivoting) If $A \in \mathbb{R}^{n\times n}$, then this algorithm computes Gauss transforms $M_1, \cdots M_{n-1}$ and interchange permutations $E_1, \cdots E_{n-1}$ such that $M_{n-1}E_{n-1}\cdots M_1E_1A = U$ is upper triangular. No multiplier is bigger than 1 in absolute value. $A(1{:}k, k)$ is overwritten by $U(1{:}k, k)$, $k = 1{:}n$. $A(k+1{:}n, k)$ is overwritten by $-M_k(k+1{:}n, k)$, $k = 1{:}n-1$. The integer vector $p(1{:}n-1)$ defines the interchange permutations. In particular, E_k interchanges rows k and $p(k)$, $k = 1{:}n-1$.

for $k = 1{:}n-1$
  Determine μ with $k \le \mu \le n$ so $|A(\mu, k)| = \| A(k{:}n, k) \|_\infty$
  $A(k, k{:}n) \leftrightarrow A(\mu, k{:}n)$
  $p(k) = \mu$
  if $A(k, k) \ne 0$
    $rows = k+1{:}n$
    $A(rows, k) = A(rows, k)/A(k, k)$
    $A(rows, rows) = A(rows, rows) - A(rows, k)A(k, rows)$
  end
end

Note that if $\| A(k{:}n, k) \|_\infty = 0$ in step k, then in exact arithmetic the first k columns of A are linearly dependent. In contrast to Algorithm 3.2.1, this poses no difficulty. We merely skip over the zero pivot.

The overhead associated with partial pivoting is minimal from the standpoint of floating point arithmetic as there are only $O(n^2)$ comparisons associated with the search for the pivots. The overall algorithm involves $2n^3/3$ flops.

To solve the linear system $Ax = b$ after invoking Algorithm 3.4.1 we

- Compute $y = M_{n-1}E_{n-1}\cdots M_1E_1b$.
- Solve the upper triangular system $Ux = y$.

All the information necessary to do this is contained in the array A and the pivot vector p. Indeed, the calculation

```
for k = 1:n - 1
    b(k) ↔ b(p(k))
    b(k + 1:n) = b(k + 1:n) - b(k)A(k + 1:n, k)
end
```

overwrites b with $M_{n-1}E_{n-1}\cdots M_1E_1$.

Example 3.4.1 If Algorithm 3.4.1 is applied to

$$A = \begin{bmatrix} 3 & 17 & 10 \\ 2 & 4 & -2 \\ 6 & 18 & -12 \end{bmatrix},$$

then upon exit

$$A = \begin{bmatrix} 6 & 18 & -12 \\ 1/3 & 8 & 16 \\ 1/2 & -1/4 & 6 \end{bmatrix}$$

and $p = [3\ ,\ 3]$. These two quantities encode all the information associated with the reduction:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1/4 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ -1/3 & 1 & 0 \\ -1/2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} A = \begin{bmatrix} 6 & 18 & -12 \\ 0 & 8 & 16 \\ 0 & 0 & 6 \end{bmatrix}.$$

3.4.4 Where is L?

Gaussian elimination with partial pivoting computes the LU factorization of a row permuted version of A. The proof is a messy subscripting argument.

Theorem 3.4.1 *If Gaussian elimination with partial pivoting is used to compute the upper triangularization*

$$M_{n-1}E_{n-1}\cdots M_1E_1A \;=\; U \qquad (3.4.1)$$

via Algorithm 3.4.1, then

$$PA = LU$$

where $P = E_{n-1}\cdots E_1$ and L is a unit lower triangular matrix with $|\ell_{ij}| \le 1$. The kth column of L below the diagonal is a permuted version of the kth Gauss vector. In particular, if $M_k = I - \tau^{(k)}e_k^T$, then $L(k+1{:}n, k) = g(k+1{:}n)$ where $g = E_{n-1}\cdots E_{k+1}\tau^{(k)}$.

Proof. A manipulation of (3.4.1) reveals that $\tilde{M}_{n-1}\cdots\tilde{M}_1PA = U$ where $\tilde{M}_{n-1} = M_{n-1}$ and

$$\tilde{M}_k \;=\; E_{n-1}\cdots E_{k+1}M_kE_{k+1}\cdots E_{n-1} \qquad k \le n-2\,.$$

Since each E_j is an interchange permutation involving row j and a row μ with $\mu \ge j$ we have $E_j(1{:}j-1, 1{:}j-1) = I_{j-1}$. It follows that each $\tilde{M}_k$ is a Gauss transform with Gauss vector $\tilde{\tau}^{(k)} \;=\; E_{n-1}\cdots E_{k+1}\tau^{(k)}$. □

As a consequence of the theorem, it is easy to see how to change Algorithm 3.4.1 so that upon completion, $A(i,j)$ houses $L(i,j)$ for all $i > j$. We merely apply each E_k to all the previously computed Gauss vectors. This is accomplished by changing the line "$A(k, k{:}n) \leftrightarrow A(\mu, k{:}n)$" in Algorithm 3.4.1 to "$A(k, 1{:}n) \leftrightarrow A(\mu, 1{:}n)$."

Example 3.4.2 The factorization $PA = LU$ of the matrix in Example 3.4.1 is given by

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 3 & 17 & 10 \\ 2 & 4 & -2 \\ 6 & 18 & -12 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1 & 0 \\ 1/3 & -1/4 & 1 \end{bmatrix} \begin{bmatrix} 6 & 18 & -12 \\ 0 & 8 & 16 \\ 0 & 0 & 6 \end{bmatrix}.$$

3.4.5 The Gaxpy Version

In §3.2 we developed outer product and gaxpy schemes for computing the LU factorization. Having just incorporated pivoting in the outer product version, it is natural to do the same with the gaxpy approach. Recall from (3.2.5) the general structure of the gaxpy LU process:

```
L = I
U = 0
for j = 1:n
    if j = 1
        v(j:n) = A(j:n, j)
    else
        Solve L(1:j - 1, 1:j - 1)z = A(1:j - 1, j) for z
            and set U(1:j - 1, j) = z.
        v(j:n) = A(j:n, j) - L(j:n, 1:j - 1)z
    end
    if j < n
        L(j + 1:n, j) = v(j + 1:n)/v(j)
    end
    U(j, j) = v(j)
end
```

With partial pivoting we search $|v(j{:}n)|$ for its maximal element and proceed accordingly. Assuming A is nonsingular so no zero pivots are encountered we obtain

```
L = I; U = 0
for j = 1:n
      if j = 1
            v(j:n) = A(j:n, j)
      else
            Solve L(1:j-1, 1:j-1)z = A(1:j-1, j)
                  for z and set U(1:j-1, j) = z.
            v(j:n) = A(j:n, j) - L(j:n, 1:j-1)z
      end                                                         (3.4.2)
      if j < n
            Determine μ with k ≤ μ ≤ n so |v(μ)| = || v(j:n) ||_∞.
            p(j) = μ
            v(j) ↔ v(μ)
            A(j, j+1:n) ↔ A(μ, j+1:n)
            L(j+1:n, j) = v(j+1:n)/v(j)
            if j > 1
                  L(j, 1:j-1) ↔ L(μ, 1:j-1)
            end
      end
      U(j, j) = v(j)
end
```

In this implementation, we emerge with the factorization $PA = LU$ where $P = E_{n-1} \cdots E_1$ where E_k is obtained by interchanging rows k and $p(k)$ of the n-by-n identity. As with Algorithm 3.4.1, this procedure requires $2n^3/3$ flops and $O(n^2)$ comparisons.

3.4.6 Error Analysis

We now examine the stability that is obtained with partial pivoting. This requires an accounting of the rounding errors that are sustained during elimination and during the triangular system solving. Bearing in mind that there are no rounding errors associated with permutation, it is not hard to show using Theorem 3.3.2 that the computed solution $\hat{x}$ satisfies $(A + E)\hat{x} = b$ where

$$|E| \le n\mathbf{u}\left(3|A| + 5\hat{P}^T|\hat{L}||\hat{U}|\right) + O(\mathbf{u}^2). \tag{3.4.3}$$

Here we are assuming that $\hat{P}$, $\hat{L}$, and $\hat{U}$ are the computed analogs of P, L, and U as produced by the above algorithms. Pivoting implies that the elements of $\hat{L}$ are bounded by one. Thus $\| \hat{L} \|_\infty \le n$ and we obtain the bound

$$\| E \|_\infty \le n\mathbf{u}\left(3\| A \|_\infty + 5n\| \hat{U} \|_\infty\right) + O(\mathbf{u}^2). \tag{3.4.4}$$

The problem now is to bound $\| \hat{U} \|_\infty$. Define the ***growth factor*** ρ by

$$\rho = \max_{i,j,k} \frac{|\hat{a}_{ij}^{(k)}|}{\| A \|_\infty} \tag{3.4.5}$$

where $\hat{A}^{(k)}$ is the computed version of the matrix $A^{(k)} = M_k E_k \cdots M_1 E_1 A$. It follows that

$$\| E \|_\infty \le 8n^3\rho\| A \|_\infty \mathbf{u} + O(\mathbf{u}^2). \tag{3.4.6}$$

Whether or not this compares favorably with the ideal bound (3.3.1) hinges upon the size of the growth factor of ρ. (The factor n^3 is not an operating factor in practice and may be ignored in this discussion.) The growth factor measures how large the numbers become during the process of elimination. In practice, ρ is usually of order 10 but it can also be as large as 2^{n-1}. Despite this, most numerical analysts regard the occurrence of serious element growth in Gaussian elimination with partial pivoting as highly unlikely in practice. The method can be used with confidence.

Example 3.4.3 If Gaussian elimination with partial pivoting is applied to the problem

$$\begin{bmatrix} .001 & 1.00 \\ 1.00 & 2.00 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1.00 \\ 3.00 \end{bmatrix}$$

with $\beta = 10$, $t = 3$, floating point arithmetic, then

$$P = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad \hat{L} = \begin{bmatrix} 1.00 & 0 \\ .001 & 1.00 \end{bmatrix}, \quad \hat{U} = \begin{bmatrix} 1.00 & 2.00 \\ 0 & 1.00 \end{bmatrix}$$

and $\hat{x} = (1.00, .996)^T$. Compare with Example 3.3.1.

Example 3.4.2 If $A \in \mathbb{R}^{n \times n}$ is defined by

$$a_{ij} = \begin{cases} 1 & \text{if } i = j \text{ or } j = n \\ -1 & \text{if } i > j \\ 0 & \text{otherwise} \end{cases}$$

then A has an LU factorization with $|\ell_{ij}| \le 1$ and $u_{nn} = 2^{n-1}$.

3.4.7 Block Gaussian Elimination

Gaussian Elimination with partial pivoting can be organized so that it is rich in level-3 operations. We detail a block outer product procedure but block gaxpy and block dot product formulations are also possible. See Dayde and Duff (1988).

Assume $A \in \mathbb{R}^{n \times n}$ and for clarity that $n = rN$. Partition A as follows:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{matrix} r \\ n-r \end{matrix} \qquad \begin{matrix} r & n-r \end{matrix}.$$

The first step in the block reduction is typical and proceeds as follows:

- Use scalar Gaussian elimination with partial pivoting (e.g. a rectangular version of Algorithm 3.4.1) to compute permutation $P_1 \in \mathbb{R}^{n\times n}$, unit lower triangular $L_{11} \in \mathbb{R}^{r\times r}$ and upper triangular $U_{11} \in \mathbb{R}^{r\times r}$ so

$$P_1 \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} = \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} U_{11}.$$

- Apply the P_1 across the rest of A:

$$\begin{bmatrix} \tilde{A}_{12} \\ \tilde{A}_{22} \end{bmatrix} = P_1 \begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix}.$$

- Solve the lower triangular multiple right hand side problem

$$L_{11}U_{12} = \tilde{A}_{12}.$$

- Perform the level-3 update

$$\tilde{A} = \tilde{A}_{22} - L_{21}U_{12}.$$

With these computations we obtain the factorization

$$P_1 A = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I_{n-r} \end{bmatrix} \begin{bmatrix} I_r & 0 \\ 0 & \tilde{A} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & I_{n-r} \end{bmatrix}.$$

The process is then repeated on the first r columns of $\tilde{A}$.

In general, during step k ($1 \le k \le N-1$) of the block algorithm we apply scalar Gaussian elimination to a matrix of size $(n-(k-1)r)$-by-r. An r-by-$(n-kr)$ multiple right hand side system is solved and a level 3 update of size $(n-kr)$-by-$(n-kr)$ is performed. The level 3 fraction for the overall process is approximately given by $1 - 3/(2N)$. Thus, for large N the procedure is rich in matrix multiplication.

3.4.8 Complete Pivoting

Another pivot strategy called *complete pivoting* has the property that the associated growth factor bound is considerably smaller than 2^{n-1}. Recall that in partial pivoting, the kth pivot is determined by scanning the current subcolumn $A(k{:}n, k)$. In complete pivoting, the largest entry in the current submatrix $A(k{:}n, k{:}n)$ is permuted into the (k,k) position. Thus, we compute the upper triangularization $M_{n-1}E_{n-1}\cdots M_1E_1AF_1\cdots F_{n-1} = U$ with the property that in step k we are confronted with the matrix

$$A^{(k-1)} = M_{k-1}E_{k-1}\cdots M_1E_1AF_1\cdots F_{k-1}$$

and determine interchange permutations E_k and F_k such that

$$\left| \left(E_kA^{(k-1)}F_k\right)_{kk} \right| = \max_{k\le i,j\le n} \left| \left(E_kA^{(k-1)}F_k\right)_{ij} \right|.$$

We have the analog of Theorem 3.4.1

Theorem 3.4.2 *If Gaussian elimination with complete pivoting is used to compute the upper triangularization*

$$M_{n-1}E_{n-1}\cdots M_1E_1AF_1\cdots F_{n-1} = U \qquad (3.4.7)$$

then

$$PAQ = LU$$

where $P = E_{n-1}\cdots E_1$, $Q = F_1\cdots F_{n-1}$ *and* L *is a unit lower triangular matrix with* $|\ell_{ij}| \le 1$. *The kth column of* L *below the diagonal is a permuted version of the kth Gauss vector. In particular, if* $M_k = I - \tau^{(k)}e_k^T$ *then* $L(k+1{:}n, k) = g(k+1{:}n)$ *where* $g = E_{n-1}\cdots E_{k+1}\tau^{(k)}$.

Proof. The proof is similar to the proof of Theorem 3.4.1. Details are left to the reader. □

Here is Gaussian elimination with complete pivoting in detail:

Algorithm 3.4.2 (Gaussian Elimination with Complete Pivoting) This algorithm computes the complete pivoting factorization $PAQ = LU$ where L is unit lower triangular and U is upper triangular. $P = E_{n-1}\cdots E_1$ and $Q = F_1\cdots F_{n-1}$ are products of interchange permutations. $A(1{:}k, k)$ is overwritten by $U(1{:}k, k), k = 1{:}n$. $A(k+1{:}n, k)$ is overwritten by $L(k+1{:}n, k), k = 1{:}n-1$. E_k interchanges rows k and $p(k)$. F_k interchanges columns k and $q(k)$.

```
for k = 1:n − 1
    Determine μ with k ≤ μ ≤ n and λ with k ≤ λ ≤ n so
        |A(μ, λ)| = max{ |A(i, j)| : i = k:n, j = k:n }
    A(k, 1:n) ↔ A(μ, 1:n)
    A(1:n, k) ↔ A(1:n, λ)
    p(k) = μ
    q(k) = λ
    if A(k, k) ≠ 0
        rows = k + 1:n
        A(rows, k) = A(rows, k)/A(k, k)
        A(rows, rows) = A(rows, rows) − A(rows, k)A(k, rows)
    end
end
```

This algorithm requires $2n^3/3$ flops and $O(n^3)$ comparisons. Unlike partial pivoting, complete pivoting involves a significant overhead because of the two-dimensional search at each stage.

3.4.9 Comments on Complete Pivoting

Suppose rank$(A) = r < n$. It follows that at the beginning of step $r+1$, $A(r+1{:}n, r+1{:}n) = 0$. This implies that $E_k = F_k = M_k = I$ for $k = r+1{:}n$ and so the algorithm can be terminated after step r with the following factorization in hand:

$$PAQ = LU = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I_{n-r} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & 0 \end{bmatrix}.$$

Here L_{11} and U_{11} are r-by-r and L_{21} and U_{12}^T are $(n-r)$-by-r. Thus, Gaussian elimination with complete pivoting can in principle be used to determine the rank of a matrix. Yet roundoff errors make the probability of encountering an exactly zero pivot remote. In practice one would have to "declare" A to have rank k if the pivot element in step $k+1$ was sufficiently small. The numerical rank determination problem is discussed in detail in §5.4.

Wilkinson (1961) has shown that in exact arithmetic the elements of the matrix $A^{(k)} = M_k E_k \cdots M_1 E_1 A F_1 \cdots F_k$ satisfy

$$|a_{ij}^{(k)}| \le k^{1/2}(2 \cdot 3^{1/2} \cdots k^{1/k-1})^{1/2} \max|a_{ij}|. \qquad (3.4.8)$$

The upper bound is a rather slow-growing function of k. This fact coupled with vast empirical evidence suggesting that ρ is always modestly sized (e.g, $\rho = 10$) permit us to conclude that *Gaussian elimination with complete pivoting is stable.* The method solves a nearby linear system $(A+E)\hat{x} = b$ exactly in the sense of (3.3.1). However, there appears to be no practical justification for choosing complete pivoting over partial pivoting except in cases where rank determination is an issue.

Example 3.4.6 If Gaussian elimination with complete pivoting is applied to the problem

$$\begin{bmatrix} .001 & 1.00 \\ 1.00 & 2.00 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1.00 \\ 3.00 \end{bmatrix}$$

with $\beta = 10, t = 3$, floating arithmetic, then

$$P = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad Q = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad \hat{L} = \begin{bmatrix} 1.00 & 0.00 \\ .500 & 1.00 \end{bmatrix}, \quad \hat{U} = \begin{bmatrix} 2.00 & 1.00 \\ 0.00 & .499 \end{bmatrix}$$

and $\hat{x} = [1.00,\ 1.00]^T$. Compare with Examples 3.3.1 and 3.4.3.

3.4.10 The Avoidance of Pivoting

For certain classes of matrices it is not necessary to pivot. It is important to identify such classes because pivoting usually degrades performance. To

illustrate the kind of analysis required to prove that pivoting can be safely avoided, we consider the case of diagonally dominant matrices. We say that $A \in \mathbb{R}^{n \times n}$ is *strictly diagonally dominant* if

$$|a_{ii}| > \sum_{\substack{j=1 \\ j \ne i}}^{n} |a_{ij}| \qquad i = 1{:}n.$$

The following theorem shows how this property can ensure a nice, no-pivoting LU factorization.

Theorem 3.4.3 *If A^T is strictly diagonally dominant, then A has an LU factorization and $|l_{ij}| \le 1$. In other words, if Algorithm 3.4.1 is applied, then $P = I$.*

Proof. Partition A as follows

$$A = \begin{bmatrix} \alpha & w^T \\ v & C \end{bmatrix}$$

where α is 1-by-1 and note that after one step of the outer product LU process we have the factorization

$$\begin{bmatrix} \alpha & w^T \\ v & C \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ v/\alpha & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & C - vw^T/\alpha \end{bmatrix} \begin{bmatrix} \alpha & w^T \\ 0 & I \end{bmatrix}.$$

The theorem follows by induction on n if we can show that the transpose of $B = C - vw^T/\alpha$ is strictly diagonally dominant. This is because we may then assume that B has an LU factorization $B = L_1 U_1$ and that implies

$$A = \begin{bmatrix} 1 & 0 \\ v/\alpha & L_1 \end{bmatrix} \begin{bmatrix} \alpha & w^T \\ 0 & U_1 \end{bmatrix} \equiv LU.$$

But the proof that B^T is strictly diagonally dominant is straight forward. From the definitions we have

$$\begin{aligned} \sum_{\substack{i=1 \\ i \ne j}}^{n-1} |b_{ij}| &= \sum_{\substack{i=1 \\ i \ne j}}^{n-1} |c_{ij} - v_i w_j/\alpha| \le \sum_{\substack{i=1 \\ i \ne j}}^{n-1} |c_{ij}| + \frac{|w_j|}{|\alpha|} \sum_{\substack{i=1 \\ i \ne j}}^{n-1} |v_i| \\ &\le (|c_{jj}| - |w_j|) + \frac{|w_j|}{|\alpha|}(|\alpha| - |v_j|) \\ &\le \left| c_{jj} - \frac{w_j v_j}{\alpha} \right| = |b_{jj}|. \square \end{aligned}$$

3.4.11 Some Applications

We conclude with some examples that illustrate how to think in terms of matrix factorizations when confronted with various linear equation situations.

Suppose A is nonsingular and n-by-n and that B is n-by-p. Consider the problem of finding X (n-by-p) so $AX = B$, i.e., the *multiple right hand side problem.* If $X = [\, x_1, \ldots, x_p \,]$ and $B = [\, b_1, \ldots, b_p \,]$ are column partitions, then

```
Compute PA = LU.
for k = 1:p
      Solve Ly = Pb_k                                  (3.4.9)
      Solve Ux_k = y
end
```

Note that A is factored just once. If $B = I_n$ then we emerge with a computed A^{-1}.

As another example of getting the LU factorization "outside the loop," suppose we want to solve the linear system $A^k x = b$ where $A \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$, and k is a positive integer. One approach is to compute $C = A^k$ and then solve $Cx = b$. However, the matrix multiplications can be avoided altogether:

```
Compute PA = LU
for j = 1:k
      Overwrite b with the solution to Ly = Pb.        (3.4.10)
      Overwrite b with the solution to Ux = b.
end
```

As a final example we show how to avoid the pitfall of explicit inverse computation. Suppose we are given $A \in \mathbb{R}^{n\times n}$, $d \in \mathbb{R}^n$, and $c \in \mathbb{R}^n$ and that we want to compute $s = c^T A^{-1} d$. One approach is to compute $X = A^{-1}$ as suggested above and then compute $s = c^T X d$. A more economical procedure is to compute $PA = LU$ and then solve the triangular systems $Ly = Pd$ and $Ux = y$. It follows that $s = c^T x$. The point of this example is to stress that when a matrix inverse is encountered in a formula, we must think in terms of solving equations rather than in terms of explicit inverse formation.

Problems

P3.4.1 Let $A = LU$ be the LU factorization of n-by-n A with $|\ell_{ij}| \le 1$. Let a_i^T and u_i^T denote the ith rows of A and U, respectively. Verify the equation

$$u_i^T = a_i^T - \sum_{j=1}^{i-1} \ell_{ij} u_j^T$$

and use it to show that $\| U \|_\infty \le 2^{n-1} \| A \|_\infty$. (Hint: Take norms and use induction.)

P3.4.2 Show that if $PAQ = LU$ is obtained via Gaussian elimination with complete pivoting, then no element of $U(i, i{:}n)$ is larger in absolute value than $|u_{ii}|$.

P3.4.3 Suppose $A \in \mathbb{R}^{n\times n}$ has an LU factorization and that L and U are known. Give an algorithm which can compute the (i, j) entry of A^{-1} in approximately $(n-j)^2 + (n-i)^2$ flops.

P3.4.4 Suppose $\hat{X}$ is the computed inverse obtained via (3.4.9). Give an upper bound for $\| A\hat{X} - I \|_F$.

P3.4.5 Prove Theorem 3.4.2.

P3.4.6 Extend Algorithm 3.4.3 so that it can factor an arbitrary rectangular matrix.

P3.4.7 Write a detailed version of the block elimination algorithm outlined in §3.4.7.

Notes and References for Sec. 3.4

An Algol version of Algorithm 3.4.1 is given in

H.J. Bowdler, R.S. Martin, G. Peters, and J.H. Wilkinson (1966). "Solution of Real and Complex Systems of Linear Equations," *Numer. Math. 8*, 217–34. See also Wilkinson and Reinsch (1971, 93–110).

The conjecture that $|a_{ij}^{(k)}| \le n \max|a_{ij}|$ when complete pivoting is used has been proven in the real $n = 4$ case in

C.W. Cryer (1968). "Pivot Size in Gaussian Elimination," *Numer. Math. 12*, 335–45.

Other papers concerned with element growth and pivoting include

J.K. Reid (1971). "A Note on the Stability of Gaussian Elimination," *J.Inst. Math. Applics. 8*, 374–75.
P.A. Businger (1971). "Monitoring the Numerical Stability of Gaussian Elimination," *Numer. Math. 16*, 360–61.
A.M. Cohen (1974). "A Note on Pivot Size in Gaussian Elimination," *Lin. Alg. and Its Applic. 8*, 361–68.
A.M. Erisman and J.K. Reid (1974). "Monitoring the Stability of the Triangular Factorization of a Sparse Matrix," *Numer. Math. 22*, 183–86.
J. Day and B. Peterson (1988). "Growth in Gaussian Elimination," *Amer. Math. Monthly 95*, 489–513.
N.J. Higham and D.J. Higham (1989). "Large Growth Factors in Gaussian Elimination with Pivoting," *SIAM J. Matrix Anal. Appl. 10*, 155–164.
L.N. Trefethen and R.S. Schreiber (1990). "Average-Case Stability of Gaussian Elimination," *SIAM J. Matrix Anal. Appl. 11*, 335–360.
N. Gould (1991). "On Growth in Gaussian Elimination with Complete Pivoting," *SIAM J. Matrix Anal. Appl. 12*, 354–361.
A. Edelman (1992). "The Complete Pivoting Conjecture for Gaussian Elimination is False," *The Mathematica Journal 2*, 58–61.
S.J. Wright (1993). "A Collection of Problems for Which Gaussian Elimination with Partial Pivoting is Unstable," *SIAM J. Sci. and Stat. Comp. 14*, 231–238.
L.V. Foster (1994). "Gaussian Elimination with Partial Pivoting Can Fail in Practice," *SIAM J. Matrix Anal. Appl. 15*, 1354–1362.

A. Edelman and W. Mascarenhas (1995). "On the Complete Pivoting Conjecture for a Hadamard Matrix of Order 12," *Linear and Multilinear Algebra 38*, 181–185.

The designers of sparse Gaussian elimination codes are interested in the topic of element growth because multipliers greater than unity are sometimes tolerated for the sake of minimizing fill-in. See

I.S. Duff, A.M. Erisman, and J.K. Reid (1986). *Direct Methods for Sparse Matrices*, Oxford University Press.

The connection between small pivots and near singularity is reviewed in

T.F. Chan (1985). "On the Existence and Computation of LU Factorizations with small pivots," *Math. Comp. 42*, 535–548.

A pivot strategy that we did not discuss is *pairwise pivoting*. In this approach, 2-by-2 Gauss transformations are used to zero the lower triangular portion of A. The technique is appealing in certain multiprocessor environments because only adjacent rows are combined in each step. See

D. Sorensen (1985). "Analysis of Pairwise Pivoting in Gaussian Elimination," *IEEE Trans. on Computers C-34*, 274–278.

As a sample paper in which a class of matrices is identified that require no pivoting, see

S. Serbin (1980). "On Factoring a Class of Complex Symmetric Matrices Without Pivoting," *Math. Comp. 35*, 1231-1234.

Just as there are six "conventional" versions of scalar Gaussian elimination, there are also six conventional block formulations of Gaussian elimination. For a discussion of these procedures and their implementation see

K. Gallivan, W. Jalby, U. Meier, and A.H. Sameh (1988). "Impact of Hierarchical Memory Systems on Linear Algebra Algorithm Design," *Int'l J. Supercomputer Applic. 2*, 12–48.

3.5 Improving and Estimating Accuracy

Suppose Gaussian elimination with partial pivoting is used to solve the n-by-n system $Ax = b$. Assume t-digit, base β floating point arithmetic is used. Equation (3.4.6) essentially says that if the growth factor is modest then the computed solution $\hat{x}$ satisfies

$$(A+E)\hat{x} = b, \qquad \| E \|_\infty \approx \mathbf{u} \| A \|_\infty, \quad \mathbf{u} = \frac{1}{2}\beta^{-t}. \tag{3.5.1}$$

In this section we explore the practical ramifications of this result. We begin by stressing the distinction that should be made between residual size and accuracy. This is followed by a discussion of scaling, iterative improvement, and condition estimation. See Higham (1996) for a more detailed treatment of these topics.

We make two notational remarks at the outset. The infinity norm is used throughout since it is very handy in roundoff error analysis and in practical

error estimation. Second, whenever we refer to "Gaussian elimination" in this section we really mean Gaussian elimination with some stabilizing pivot strategy such as partial pivoting.

3.5.1 Residual Size Versus Accuracy

The *residual* of a computed solution $\hat{x}$ to the linear system $Ax = b$ is the vector $b - A\hat{x}$. A small residual means that $A\hat{x}$ effectively "predicts" the right hand side b. From (3.5.1) we have $\| b - A\hat{x} \|_\infty \approx \mathbf{u} \| A \|_\infty \| \hat{x} \|_\infty$ and so we obtain

Heuristic I. Gaussian elimination produces a solution $\hat{x}$ with a relatively small residual.

Small residuals do not imply high accuracy. Combining (3.3.2) and (3.5.1), we see that

$$\frac{\| \hat{x} - x \|_\infty}{\| x \|_\infty} \approx \mathbf{u}\kappa_\infty(A)\,. \tag{3.5.2}$$

This justifies a second guiding principle.

Heuristic II. If the unit roundoff and condition satisfy $\mathbf{u} \approx 10^{-d}$ and $\kappa_\infty(A) \approx 10^q$, then Gaussian elimination produces a solution $\hat{x}$ that has about $d - q$ correct decimal digits.

If $\mathbf{u}\kappa_\infty(A)$ is large, then we say that A is ill-conditioned with respect to the machine precision.

As an illustration of the Heuristics I and II, consider the system

$$\begin{bmatrix} .986 & .579 \\ .409 & .237 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} .235 \\ .107 \end{bmatrix}$$

in which $\kappa_\infty(A) \approx 700$ and $x = (2, -3)^T$. Here is what we find for various machine precisions:

β	t	$\hat{x}_1$	$\hat{x}_2$	$\frac{\| \hat{x} - x \|_\infty}{\| x \|_\infty}$	$\frac{\| b - A\hat{x} \|_\infty}{\| A \|_\infty \| \hat{x} \|_\infty}$
10	3	2.11	-3.17	$5 \cdot 10^{-2}$	$2.0 \cdot 10^{-3}$
10	4	1.986	-2.975	$8 \cdot 10^{-3}$	$1.5 \cdot 10^{-4}$
10	5	2.0019	-3.0032	$1 \cdot 10^{-3}$	$2.1 \cdot 10^{-6}$
10	6	2.00025	-3.00094	$3 \cdot 10^{-4}$	$4.2 \cdot 10^{-7}$

Whether or not one is content with the computed solution $\hat{x}$ depends on the requirements of the underlying source problem. In many applications accuracy is not important but small residuals are. In such a situation, the $\hat{x}$ produced by Gaussian elimination is probably adequate. On the other hand, if the number of correct digits in $\hat{x}$ is an issue then the situation is more complicated and the discussion in the remainder of this section is relevant.

3.5.2 Scaling

Let β be the machine base and define the diagonal matrices D_1 and D_2 by

$$\begin{aligned} D_1 &= \text{diag}(\beta^{r_1} \ldots \beta^{r_n}) \\ D_2 &= \text{diag}(\beta^{c_1} \ldots \beta^{c_n}). \end{aligned}$$

The solution to the n-by-n linear system $Ax = b$ can be found by solving the *scaled system* $(D_1^{-1}AD_2)y = D_1^{-1}b$ using Gaussian elimination and then setting $x = D_2y$. The scalings of A, b, and y require only $O(n^2)$ flops and may be accomplished without roundoff. Note that D_1 scales equations and D_2 scales unknowns.

It follows from Heuristic II that if $\hat{x}$ and $\hat{y}$ are the computed versions of x and y, then

$$\frac{\| D_2^{-1}(\hat{x} - x) \|_\infty}{\| D_2^{-1}x \|_\infty} = \frac{\| \hat{y} - y \|_\infty}{\| y \|_\infty} \approx \mathbf{u}\kappa_\infty(D_1^{-1}AD_2). \tag{3.5.3}$$

Thus, if $\kappa_\infty(D_1^{-1}AD_2)$ can be made considerably smaller than $\kappa_\infty(A)$, then we might expect a correspondingly more accurate $\hat{x}$, provided errors are measured in the "D_2" norm defined by $\| z \|_{D_2} = \| D_2^{-1}z \|_\infty$. This is the objective of scaling. Note that it encompasses two issues: the condition of the scaled problem and the appropriateness of appraising error in the D_2-norm.

An interesting but very difficult mathematical problem concerns the exact minimization of $\kappa_p(D_1^{-1}AD_2)$ for general diagonal D_i and various p. What results there are in this direction are not very practical. This is hardly discouraging, however, when we recall that (3.5.3) is heuristic and it makes little sense to minimize exactly a heuristic bound. What we seek is a fast, approximate method for improving the quality of the computed solution $\hat{x}$.

One technique of this variety is *simple row scaling*. In this scheme D_2 is the identity and D_1 is chosen so that each row in $D_1^{-1}A$ has approximately the same ∞-norm. Row scaling reduces the likelihood of adding a very small number to a very large number during elimination—an event that can greatly diminish accuracy.

Slightly more complicated than simple row scaling is *row-column equilibration*. Here, the object is to choose D_1 and D_2 so that the ∞-norm of each row and column of $D_1^{-1}AD_2$ belongs to the interval $[1/\beta, 1]$ where β is the base of the floating point system. For work along these lines see McKeeman (1962).

It cannot be stressed too much that simple row scaling and row-column equilibration do not "solve" the scaling problem. Indeed, either technique can render a worse $\hat{x}$ than if no scaling whatever is used. The ramifications of this point are thoroughly discussed in Forsythe and Moler (1967, chapter 11). The basic recommendation is that the scaling of equations and

unknowns must proceed on a problem-by-problem basis. General scaling strategies are unreliable. It is best to scale (if at all) on the basis of what the source problem proclaims about the significance of each a_{ij}. Measurement units and data error may have to be considered.

Example 3.5.1 (Forsythe and Moler (1967, pp. 34, 40)). If

$$\begin{bmatrix} 10 & 100,000 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 100,000 \\ 2 \end{bmatrix}$$

and the equivalent row-scaled problem

$$\begin{bmatrix} .0001 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

are each solved using $\beta = 10, t = 3$ arithmetic, then solutions $\hat{x} = (0.00,\ 1.00)^T$ and $\hat{x} = (1.00,\ 1.00)^T$ are respectively computed. Note that $x = (1.0001\ldots,\ .9999\ldots)^T$ is the exact solution.

3.5.3 Iterative Improvement

Suppose $Ax = b$ has been solved via the partial pivoting factorization $PA = LU$ and that we wish to improve the accuracy of the computed solution $\hat{x}$. If we execute

$$\begin{aligned} & r = b - A\hat{x} \\ & \text{Solve } Ly = Pr. \\ & \text{Solve } Uz = y. \\ & x_{new} = \hat{x} + z \end{aligned} \tag{3.5.4}$$

then in exact arithmetic $Ax_{new} = A\hat{x} + Az = (b - r) + r = b$. Unfortunately, the naive floating point execution of these formulae renders an x_{new} that is no more accurate than $\hat{x}$. This is to be expected since $\hat{r} = fl(b - A\hat{x})$ has few, if any, correct significant digits. (Recall Heuristic I.) Consequently, $\hat{z} = fl(A^{-1}r) \approx A^{-1} \cdot \text{noise} \approx \text{noise}$ is a very poor correction *from the standpoint of improving the accuracy of* $\hat{x}$. However, Skeel (1980) has done an error analysis that indicates when (3.5.4) gives an improved x_{new} *from the standpoint of backwards error*. In particular, if the quantity

$$\tau = \left(\| \, |A| \, |A^{-1}| \, \|_\infty\right) \left(\max_i \ (|A||x|)_i \,/ \min_i \ (|A||x|)_i\right)$$

is not too big, then (3.5.4) produces an x_{new} such that $(A + E)x_{new} = b$ for very small E. Of course, if Gaussian elimination with partial pivoting is used then the computed $\hat{x}$ already solves a nearby system. However, this may not be the case for some of the pivot strategies that are used to preserve sparsity. In this situation, the *fixed precision iterative improvement*

step (3.5.4) can be very worthwhile and cheap. See Arioli, Demmel, and Duff (1988).

For (3.5.4) to produce a more accurate x, it is necessary to compute the residual $b - A\hat{x}$ with extended precision floating point arithmetic. Typically, this means that if t-digit arithmetic is used to compute $PA = LU$, x, y, and z, then $2t$-digit arithmetic is used to form $b - A\hat{x}$, i.e., double precision. The process can be iterated. In particular, once we have computed $PA = LU$ and initialize $x = 0$, we repeat the following:

$$\begin{aligned} & r = b - Ax \text{ (Double Precision)} \\ & \text{Solve } Ly = Pr \text{ for } y. \\ & \text{Solve } Uz = y \text{ for } z. \\ & x = x + z \end{aligned} \qquad (3.5.5)$$

We refer to this process as *mixed precision iterative improvement*. The original A must be used in the double precision computation of r. The basic result concerning the performance of (3.5.5) is summarized in the following heuristic:

Heuristic III. If the machine precision u and condition satisfy $\mathbf{u} = 10^{-d}$ and $\kappa_\infty(A) \approx 10^q$, then after k executions of (3.5.5), x has approximately $\min(d, k(d-q))$ correct digits.

Roughly speaking, if $\mathbf{u}\kappa_\infty(A) \leq 1$, then iterative improvement can ultimately produce a solution that is correct to full (single) precision. Note that the process is relatively cheap. Each improvement costs $O(n^2)$, to be compared with the original $O(n^3)$ investment in the factorization $PA = LU$. Of course, no improvement may result if A is badly enough conditioned with respect to the machine precision.

The primary drawback of mixed precision iterative improvement is that its implementation is somewhat machine-dependent. This discourages its use in software that is intended for wide distribution. The need for retaining an original copy of A is another aggravation associated with the method.

On the other hand, mixed precision iterative improvement is usually very easy to implement on a given machine that has provision for the accumulation of inner products, i.e., provision for the double precision calculation of inner products between the rows of A and x. In a short mantissa computing environment the presence of an iterative improvement routine can significantly widen the class of solvable $Ax = b$ problems.

Example 3.5.2 If (3.5.5) is applied to the system

$$\begin{bmatrix} .986 & .579 \\ .409 & .237 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} .235 \\ .107 \end{bmatrix}$$

and $\beta = 10$ and $t = 3$, then iterative improvement produces the following sequence of computed solutions:

$$\hat{x} = \begin{bmatrix} 2.11 \\ -3.17 \end{bmatrix}, \begin{bmatrix} 1.99 \\ -2.99 \end{bmatrix}, \begin{bmatrix} 2.00 \\ -3.00 \end{bmatrix}, \ldots$$

The exact solution is $x = [2, -3]^T$.

3.5.4 Condition Estimation

Suppose that we have solved $Ax = b$ via $PA = LU$ and that we now wish to ascertain the number of correct digits in the computed solution $\hat{x}$. It follows from Heuristic II that in order to do this we need an estimate of the condition $\kappa_\infty(A) = \| A \|_\infty \| A^{-1} \|_\infty$. Computing $\| A \|_\infty$ poses no problem as we merely use the formula

$$\| A \|_\infty = \max_{1 \leq i \leq n} \sum_{j=1}^{n} |a_{ij}|.$$

The challenge is with respect to the factor $\| A^{-1} \|_\infty$. Conceivably, we could estimate this quantity by $\| \hat{X} \|_\infty$, where $\hat{X} = [\, \hat{x}_1, \ldots, \hat{x}_n \,]$ and $\hat{x}_i$ is the computed solution to $Ax_i = e_i$. (See §3.4.9.) The trouble with this approach is its expense: $\hat{\kappa}_\infty = \| A \|_\infty \| \hat{X} \|_\infty$ costs about three times as much as $\hat{x}$.

The central problem of *condition estimation* is how to estimate the condition number in $O(n^2)$ flops assuming the availability of $PA = LU$ or some other factorizations that are presented in subsequent chapters. An approach described in Forsythe and Moler (SLE, p. 51) is based on iterative improvement and the heuristic $\mathbf{u}\kappa_\infty(A) \approx \| z \|_\infty / \| x \|_\infty$ where z is the first correction of x in (3.5.5). While the resulting condition estimator is $O(n^2)$, it suffers from the shortcoming of iterative improvement, namely, machine dependency.

Cline, Moler, Stewart, and Wilkinson (1979) have proposed a very successful approach to the condition estimation problem without this flaw. It is based on exploitation of the implication

$$Ay = d \quad \Longrightarrow \quad \| A^{-1} \|_\infty \geq \| y \|_\infty / \| d \|_\infty.$$

The idea behind their estimator is to choose d so that the solution y is large in norm and then set

$$\hat{\kappa}_\infty = \| A \|_\infty \| y \|_\infty / \| d \|_\infty.$$

The success of this method hinges on how close the ratio $\| y \|_\infty / \| d \|_\infty$ is to its maximum value $\| A^{-1} \|_\infty$.

Consider the case when $A = T$ is upper triangular. The relation between d and y is completely specified by the following column version of back substitution:

$$p(1{:}n) = 0$$

for $k = n{:} -1{:}1$
  Choose $d(k)$.
  $y(k) = (d(k) - p(k))/T(k,k)$ (3.5.6)
  $p(1{:}k-1) = p(1{:}k-1) + y(k)T(1{:}k-1,k)$
end

Normally, we use this algorithm to solve a *given* triangular system $Ty = d$. Now, however, we are free to pick the right-hand side d subject to the "constraint" that y is large relative to d.

One way to encourage growth in y is to choose $d(k)$ from the set $\{-1, +1\}$ so as to maximize $y(k)$. If $p(k) \geq 0$, then set $d(k) = -1$. If $p(k) < 0$, then set $d(k) = +1$. In other words, (3.5.6) is invoked with $d(k)$ = -sign($p(k)$). Since d is then a vector of the form $d(1{:}n) = (\pm 1, \ldots, \pm 1)^T$, we obtain the estimator $\hat{\kappa}_\infty = \| T \|_\infty \| y \|_\infty$.

A more reliable estimator results if $d(k) \in \{-1, +1\}$ is chosen so as to encourage growth both in $y(k)$ and the updated running sum given by $p(1{:}k-1,k) + T(1{:}k-1,k)y(k)$. In particular, at step k we compute

$$y(k)^+ = (1 - p(k))/T(k,k)$$

$$s(k)^+ = |y(k)^+| + \| p(1{:}k-1) + T(1{:}k-1,k)y(k)^+ \|_1$$

$$y(k)^- = (-1 - p(k))/T(k,k)$$

$$s(k)^- = |y(k)^-| + \| p(1{:}k-1) + T(1{:}k-1,k)y(k)^- \|_1$$

and set

$$y(k) = \begin{cases} y(k)^+ & \text{if } s(k)^+ \geq s(k)^- \\ y(k)^- & \text{if } s(k)^+ < s(k)^- \end{cases}$$

This gives

Algorithm 3.5.1 (Condition Estimator) Let $T \in \mathbb{R}^{n \times n}$ be a nonsingular upper triangular matrix. This algorithm computes unit ∞-norm y and a scalar κ so $\| Ty \|_\infty \approx 1/\| T^{-1} \|_\infty$ and $\kappa \approx \kappa_\infty(T)$

$p(1{:}n) = 0$
for $k = n{:} -1{:}1$
  $y(k)^+ = (1 - p(k))/T(k,k)$
  $y(k)^- = (-1 - p(k))/T(k,k)$
  $p(k)^+ = p(1{:}k-1) + T(1{:}k-1,k)y(k)^+$
  $p(k)^- = p(1{:}k-1) + T(1{:}k-1,k)y(k)^-$

  if $|y(k)^+| + \| p(k)^+ \|_1 \geq |y(k)^-| + \| p(k)^- \|_1$
    $y(k) = y(k)^+$
    $p(1{:}k-1) = p(k)^+$
  else
    $y(k) = y(k)^-$
    $p(1{:}k-1) = p(k)^-$
  end
end
$\kappa = \| y \|_\infty \| T \|_\infty$;
$y = y / \| y \|_\infty$

The algorithm involves several times the work of ordinary back substitution.

We are now in a position to describe a procedure for estimating the condition of a square nonsingular matrix A whose $PA = LU$ factorization we know:

- Apply the lower triangular version of Algorithm 3.5.1 to U^T and obtain a large norm solution to $U^T y = d$.
- Solve the triangular systems $L^T r = y$, $Lw = Pr$, and $Uz = w$.
- $\hat{\kappa}_\infty = \| A \|_\infty \| z \|_\infty / \| r \|_\infty$.

Note that $\| z \|_\infty \leq \| A^{-1} \|_\infty \| r \|_\infty$. The method is based on several heuristics. First, if A is ill-conditioned and $PA = LU$, then it is usually the case that U is correspondingly ill-conditioned. The lower triangle L tends to be fairly well-conditioned. Thus, it is more profitable to apply the condition estimator to U than to L. The vector r, because it solves $A^T P^T r = d$, tends to be rich in the direction of the left singular vector associated with $\sigma_{min}(A)$. Righthand sides with this property render large solutions to the problem $Az = r$.

In practice, it is found that the condition estimation technique that we have outlined produces good order-of-magnitude estimates of the actual condition number.

Problems

P3.5.1 Show by example that there may be more than one way to equilibrate a matrix.

P3.5.2 Using $\beta = 10$, $t = 2$ arithmetic, solve

$$\begin{bmatrix} 11 & 15 \\ 5 & 7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 7 \\ 3 \end{bmatrix}$$

using Gaussian elimination with partial pivoting. Do one step of iterative improvement using $t = 4$ arithmetic to compute the residual. (Do not forget to round the computed residual to two digits.)

P3.5.3 Suppose $P(A + E) = \hat{L}\hat{U}$, where P is a permutation, $\hat{L}$ is lower triangular with $|\hat{\ell}_{ij}| \leq 1$, and $\hat{U}$ is upper triangular. Show that $\hat{\kappa}_\infty(A) \geq \| A \|_\infty / (\| E \|_\infty + \mu)$ where

$\mu = \min |\hat{u}_{ii}|$. Conclude that if a small pivot is encountered when Gaussian elimination with pivoting is applied to A, then A is ill-conditioned. The converse is not true. (Let $A = B_n$).

P3.5.4 (Kahan 1966) The system $Ax = b$ where

$$A = \begin{bmatrix} 2 & -1 & 1 \\ -1 & 10^{-10} & 10^{-10} \\ 1 & 10^{-10} & 10^{-10} \end{bmatrix} \qquad b = \begin{bmatrix} 2(1+10^{-10}) \\ -10^{-10} \\ 10^{-10} \end{bmatrix}$$

has solution $x = (10^{-10}\ -1\ 1)^T$. (a) Show that if $(A+E)y = b$ and $|E| \leq 10^{-8}|A|$, then $|x - y| \leq 10^{-7}|x|$. That is, small relative changes in A's entries do not induce large changes in x even though $\kappa_\infty(A) = 10^{10}$. (b) Define $D = \text{diag}(10^{-5}, 10^5, 10^5)$. Show $\kappa_\infty(DAD) \leq 5$. (c) Explain what is going on in terms of Theorem 2.7.3.

P3.5.5 Consider the matrix:

$$T = \begin{bmatrix} 1 & 0 & M & -M \\ 0 & 1 & -M & M \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad M \in \mathbb{R}.$$

What estimate of $\kappa_\infty(T)$ is produced when (3.5.6) is applied with $d(k) = -\text{sign}(p(k))$? What estimate does Algorithm 3.5.1 produce? What is the true $\kappa_\infty(T)$?

P3.5.6 What does Algorithm 3.5.1 produce when applied to the matrix B_n given in (2.7.9)?

Notes and References for Sec. 3.5

The following papers are concerned with the scaling of $Ax = b$ problems:

F.L. Bauer (1963). "Optimally Scaled Matrices," *Numer. Math. 5*, 73–87.

P.A. Businger (1968). "Matrices Which Can be Optimally Scaled," *Numer. Math. 12*, 346–48.

A. van der Sluis (1969). "Condition Numbers and Equilibration Matrices," *Numer. Math. 14*, 14–23.

A. van der Sluis (1970). "Condition, Equilibration, and Pivoting in Linear Algebraic Systems," *Numer. Math. 15*, 74–86.

C. McCarthy and G. Strang (1973). "Optimal Conditioning of Matrices," *SIAM J. Num. Anal. 10*, 370–88.

T. Fenner and G. Loizou (1974). "Some New Bounds on the Condition Numbers of Optimally Scaled Matrices," *J. ACM 21*, 514–24.

G.H. Golub and J.M. Varah (1974). "On a Characterization of the Best L_2-Scaling of a Matrix," *SIAM J. Num. Anal. 11*, 472–79.

R. Skeel (1979). "Scaling for Numerical Stability in Gaussian Elimination," *J. ACM. 26*, 494–526.

R. Skeel (1981). "Effect of Equilibration on Residual Size for Partial Pivoting," *SIAM J. Num. Anal. 18*, 449–55.

Part of the difficulty in scaling concerns the selection of a norm in which to measure errors. An interesting discussion of this frequently overlooked point appears in

W. Kahan (1966). "Numerical Linear Algebra," *Canadian Math. Bull. 9*, 757–801.

For a rigorous analysis of iterative improvement and related matters, see

M. Jankowski and M. Wozniakowski (1977). "Iterative Refinement Implies Numerical Stability," *BIT 17*, 303–311.

C.B. Moler (1967). "Iterative Refinement in Floating Point," *J. ACM 14*, 316–71.

R.D. Skeel (1980). "Iterative Refinement Implies Numerical Stability for Gaussian Elimination," *Math. Comp. 35*, 817–832.

G.W. Stewart (1981). "On the Implicit Deflation of Nearly Singular Systems of Linear Equations," *SIAM J. Sci. and Stat. Comp. 2*, 136-140.

The condition estimator that we described is given in

A.K. Cline, C.B. Moler, G.W. Stewart, and J.H. Wilkinson (1979). "An Estimate for the Condition Number of a Matrix," *SIAM J. Num. Anal.* 16, 368-75.

Other references concerned with the condition estimation problem include

C.G. Broyden (1973). "Some Condition Number Bounds for the Gaussian Elimination Process," *J. Inst. Math. Applic. 12*, 273–86.

F. Lemeire (1973). "Bounds for Condition Numbers of Triangular Value of a Matrix," *Lin. Alg. and Its Applic. 11*, 1–2.

R.S. Varga (1976). "On Diagonal Dominance Arguments for Bounding $\| A^{-1} \|_\infty$," *Lin. Alg. and Its Applic. 14*, 211–17.

G.W. Stewart (1980). "The Efficient Generation of Random Orthogonal Matrices with an Application to Condition Estimators," *SIAM J. Num. Anal. 17*, 403–9.

D.P. O'Leary (1980). "Estimating Matrix Condition Numbers," *SIAM J. Sci. Stat. Comp. 1*, 205–9.

R.G. Grimes and J.G. Lewis (1981). "Condition Number Estimation for Sparse Matrices," *SIAM J. Sci. and Stat. Comp. 2*, 384–88.

A.K. Cline, A.R. Conn, and C. Van Loan (1982). "Generalizing the LINPACK Condition Estimator," in *Numerical Analysis* , ed., J.P. Hennart, Lecture Notes in Mathematics no. 909, Springer-Verlag, New York.

A.K. Cline and R.K. Rew (1983). "A Set of Counter examples to Three Condition Number Estimators," *SIAM J. Sci. and Stat. Comp. 4*, 602–611.

W. Hager (1984). "Condition Estimates," *SIAM J. Sci. and Stat. Comp. 5*, 311–316.

N.J. Higham (1987). "A Survey of Condition Number Estimation for Triangular Matrices," *SIAM Review 29*, 575–596.

N.J. Higham (1988). "Fortran Codes for Estimating the One-norm of a Real or Complex Matrix, with Applications to Condition Estimation," *ACM Trans. Math. Soft. 14*, 381–396.

N.J. Higham (1988). "FORTRAN Codes for Estimating the One-Norm of a Real or Complex Matrix with Applications to Condition Estimation (Algorithm 674)," *ACM Trans. Math. Soft. 14*, 381–396.

C.H. Bischof (1990). "Incremental Condition Estimation," *SIAM J. Matrix Anal. Appl. 11*, 644–659.

C.H. Bischof (1990). "Incremental Condition Estimation for Sparse Matrices," *SIAM J. Matrix Anal. Appl. 11*, 312–322.

G. Auchmuty (1991). "A Posteriori Error Estimates for Linear Equations," *Numer. Math. 61*, 1–6.

N.J. Higham (1993). "Optimization by Direct Search in Matrix Computations," *SIAM J. Matrix Anal. Appl. 14*, 317–333.

D.J. Higham (1995). "Condition Numbers and Their Condition Numbers," *Lin. Alg. and Its Applic. 214*, 193–213.

Chapter 4

Special Linear Systems

§4.1 The LDM^T and LDL^T Factorizations
§4.2 Positive Definite Systems
§4.3 Banded Systems
§4.4 Symmetric Indefinite Systems
§4.5 Block Systems
§4.6 Vandermonde Systems and the FFT
§4.7 Toeplitz and Related Systems

It is a basic tenet of numerical analysis that structure should be exploited whenever solving a problem. In numerical linear algebra, this translates into an expectation that algorithms for general matrix problems can be streamlined in the presence of such properties as symmetry, definiteness, and sparsity. This is the central theme of the current chapter, where our principal aim is to devise special algorithms for computing special variants of the LU factorization.

We begin by pointing out the connection between the triangular factors L and U when A is symmetric. This is achieved by examining the LDM^T factorization in §4.1. We then turn our attention to the important case when A is both symmetric and positive definite, deriving the stable Cholesky factorization in §4.2. Unsymmetric positive definite systems are also investigated in this section. In §4.3, banded versions of Gaussian elimination and other factorization methods are discussed. We then examine the interesting situation when A is symmetric but indefinite. Our treatment of this problem in §4.4 highlights the numerical analyst's ambivalence towards pivoting. We love pivoting for the stability it induces but despise it for the

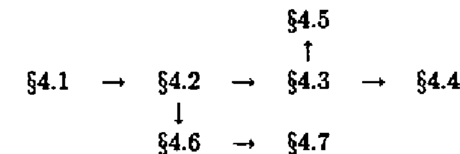
structure that it can destroy. Fortunately, there is a happy resolution to this conflict in the symmetric indefinite problem.

Any block banded matrix is also banded and so the methods of §4.3 are applicable. Yet, there are occasions when it pays not to adopt this point of view. To illustrate this we consider the important case of block tridiagonal systems in §4.5. Other block systems are discussed as well.

In the final two sections we examine some very interesting $O(n^2)$ algorithms that can be used to solve Vandermonde and Toeplitz systems.

Before You Begin

Chapter 1, §§2.1-2.5, and §2.7, and Chapter 3 are assumed. Within this chapter there are the following dependencies:



Complementary references include George and Liu (1981), Gill, Murray, and Wright (1991), Higham (1996), Trefethen and Bau (1996), and Demmel (1996). Some MATLAB functions important to this chapter: `chol`, `tril`, `triu`, `vander`, `toeplitz`, `fft`. LAPACK connections include

LAPACK: General Band Matrices	
`_GBSV`	Solve $AX = B$
`_CGBCON`	Condition estimator
`_GBRFS`	Improve $AX = B$, $A^TX = B$, $A^HX = B$ solutions with error bounds
`_GBSVX`	Solve $AX = B$, $A^TX = B$, $A^HX = B$ with condition estimate
`_GBTRF`	$PA = LU$
`_GBTRS`	Solve $AX = B$, $A^TX = B$, $A^HX = B$ via $PA = LU$
`_GBEQU`	Equilibration

LAPACK: General Tridiagonal Matrices	
`_GTSV`	Solve $AX = B$
`_GTCON`	Condition estimator
`_GTRFS`	Improve $AX = B$, $A^TX = B$, $A^HX = B$ solutions with error bounds
`_GTSVX`	Solve $AX = B$, $A^TX = B$, $A^HX = B$ with condition estimate
`_GTTRF`	$PA = LU$
`_GTTRS`	Solve $AX = B$, $A^TX = B$, $A^HX = B$ via $PA = LU$

LAPACK: Full Symmetric Positive Definite	
`_POSV`	Solve $AX = B$
`_POCON`	Condition estimate via $PA = LU$
`_PORFS`	Improve $AX = B$ solutions with error bounds
`_POSVX`	Solve $AX = B$ with condition estimate
`_POTRF`	$A = GG^T$
`_POTRS`	Solve $AX = B$ via $A = GG^T$
`_POTRI`	A^{-1}
`_POEQU`	Equilibration

LAPACK: Banded Symmetric Positive Definite	
_PBSV	Solve $AX = B$
_PBCON	Condition estimate via $A = GG^T$
_PBRFS	Improve $AX = B$ solutions with error bounds
_PBSVX	Solve $AX = B$ with condition estimate
_PBTRF	$A = GG^T$
_PBTRS	Solve $AX = B$ via $A = GG^T$

LAPACK: Tridiagonal Symmetric Positive Definite	
_PTSV	Solve $AX = B$
_PTCON	Condition estimate via $A = LDL^T$
_PTRFS	Improve $AX = B$ solutions with error bounds
_PTSVX	Solve $AX = B$ with condition estimate
_PTTRF	$A = LDL^T$
_PTTRS	Solve $AX = B$ via $A = LDL^T$

LAPACK: Full Symmetric Indefinite	
_SYSV	Solve $AX = B$
_SYCON	Condition estimate via $PAP^T = LDL^T$
_SYRFS	Improve $AX = B$ solutions with error bounds
_SYSVX	Solve $AX = B$ with condition estimate
_SYTRF	$PAP^T = LDL^T$
_SYTRS	Solve $AX = B$ via $PAP^T = LDL^T$
_SYTRI	A^{-1}

LAPACK: Triangular Banded Matrices	
_TBCON	Condition estimate
_TBRFS	Improve $AX = B$, $A^TX = B$ solutions with error bounds
_TBTRS	Solve $AX = B$, $A^TX = B$

4.1 The LDM^T and LDL^T Factorizations

We want to develop a structure-exploiting method for solving symmetric $Ax = b$ problems. To do this we establish a variant of the LU factorization in which A is factored into a three-matrix product LDM^T where D is diagonal and L and M are unit lower triangular. Once this factorization is obtained, the solution to $Ax = b$ may be found in $O(n^2)$ flops by solving $Ly = b$ (forward elimination), $Dz = y$, and $M^Tx = z$ (back substitution). The reason for developing the LDM^T factorization is to set the stage for the symmetric case for if $A = A^T$ then $L = M$ and the work associated with the factorization is half of that required by Gaussian elimination. The issue of pivoting is taken up in subsequent sections.

4.1.1 The LDM^T Factorization

Our first result connects the LDM^T factorization with the LU factorization.

Theorem 4.1.1 *If all the leading principal submatrices of $A \in \mathbb{R}^{n\times n}$ are nonsingular, then there exist unique unit lower triangular matrices L and M and a unique diagonal matrix $D = \text{diag}(d_1, \ldots, d_n)$ such that $A = LDM^T$.*

Proof. By Theorem 3.2.1 we know that A has an LU factorization $A = LU$. Set $D = \text{diag}(d_1, \ldots, d_n)$ with $d_i = u_{ii}$ for $i = 1{:}n$. Notice that D is nonsingular and that $M^T = D^{-1}U$ is unit upper triangular. Thus, $A = LU = LD(D^{-1}U) = LDM^T$. Uniqueness follows from the uniqueness of the LU factorization as described in Theorem 3.2.1. □

The proof shows that the LDM^T factorization can be found by using Gaussian elimination to compute $A = LU$ and then determining D and M from the equation $U = DM^T$. However, an interesting alternative algorithm can be derived by computing L, D, and M directly.

Assume that we know the first $j-1$ columns of L, diagonal entries $d_1, \ldots, d_{j-1}$ of D, and the first $j-1$ rows of M for some j with $1 \le j \le n$. To develop recipes for $L(j+1{:}n, j)$, $M(j, 1{:}j-1)$, and d_j we equate jth columns in the equation $A = LDM^T$. In particular,

$$A(1{:}n, j) = Lv \tag{4.1.1}$$

where $v = DM^Te_j$. The "top" half of (4.1.1) defines $v(1{:}j)$ as the solution of a known lower triangular system:

$$L(1{:}j, 1{:}j)v(1{:}j) = A(1{:}j, j).$$

Once we know v then we compute

$$\begin{aligned} d(j) &= v(j) \\ M(j,i) &= v(i)/d(i) \qquad i = 1{:}j-1. \end{aligned}$$

The "bottom" half of (4.1.1) says $L(j+1{:}n, 1{:}j)v(1{:}j) = A(j+1{:}n, j)$ which can be rearranged to obtain a recipe for the jth column of L:

$$L(j+1{:}n, j)v(j) = A(j+1{:}n, j) - L(j+1{:}n, 1{:}j-1)v(1{:}j-1).$$

Thus, $L(j+1{:}n, j)$ is a scaled gaxpy operation and overall we obtain

```
for j = 1:n
    Solve L(1:j,1:j)v(1:j) = A(1:j,j) for v(1:j).
    for i = 1:j-1
        M(j,i) = v(i)/d(i)                                    (4.1.2)
    end
    d(j) = v(j)
    L(j+1:n,j) =
        (A(j+1:n,j) - L(j+1:n,1:j-1)v(1:j-1)) /v(j)
end
```

As with the LU factorization, it is possible to overwrite A with the L, D, and M factors. If the column version of forward elimination is used to solve for $v(1{:}j)$ then we obtain the following procedure:

Algorithm 4.1.1 (LDM^T) If $A \in \mathbb{R}^{n\times n}$ has an LU factorization then this algorithm computes unit lower triangular matrices L and M and a diagonal matrix $D = \text{diag}(d_1, \ldots, d_n)$ such that $A = LDM^T$. The entry a_{ij} is overwritten with ℓ_{ij} if $i > j$, with d_i if $i = j$, and with m_{ji} if $i < j$.

```
for j = 1:n
    { Solve L(1:j,1:j)v(1:j) = A(1:j,j). }
    v(1:j) = A(1:j,j)
    for k = 1:j-1
        v(k+1:j) = v(k+1:j) - v(k)A(k+1:j,k)
    end
    { Compute M(j,1:j-1) and store in A(1:j-1,j). }
    for i = 1:j-1
        A(i,j) = v(i)/A(i,i)
    end
    { Store d(j) in A(j,j). }
    A(j,j) = v(j)
    { Compute L(j+1:n,j) and store in A(j+1:n,j) }
    for k = 1:j-1
        A(j+1:n) = A(j+1:n,j) - v(k)A(j+1:n,k)
    end
    A(j+1:n,j) = A(j+1:n,j)/v(j)
end
```

This algorithm involves the same amount of work as the LU factorization, about $2n^3/3$ flops.

The computed solution $\hat{x}$ to $Ax = b$ obtained via Algorithm 4.1.1 and the usual triangular system solvers of §3.1 can be shown to satisfy a perturbed system $(A + E)\hat{x} = b$, where

$$|E| \le n\mathbf{u}\left(3|A| + 5|\hat{L}||\hat{D}||\hat{M}^T|\right) + O(\mathbf{u}^2) \tag{4.1.3}$$

and $\hat{L}$, $\hat{D}$, and $\hat{M}$ are the computed versions of L, D, and M, respectively.

As in the case of the LU factorization considered in the previous chapter, the upper bound in (4.1.3) is without limit unless some form of pivoting is done. Hence, for Algorithm 4.1.1 to be a practical procedure, it must be modified so as to compute a factorization of the form $PA = LDM^T$, where P is a permutation matrix chosen so that the entries in L satisfy $|\ell_{ij}| \le 1$. The details of this are not pursued here since they are straightforward and since our main object for introducing the LDM^T factorization is to motivate

special methods for symmetric systems.

Example 4.1.1

$$A = \begin{bmatrix} 10 & 10 & 20 \\ 20 & 25 & 40 \\ 30 & 50 & 61 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 4 & 1 \end{bmatrix} \begin{bmatrix} 10 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and upon completion, Algorithm 4.1.1 overwrites A as follows:

$$A = \begin{bmatrix} 10 & 1 & 2 \\ 2 & 5 & 0 \\ 3 & 4 & 1 \end{bmatrix}.$$

4.1.2 Symmetry and the LDL^T Factorization

There is redundancy in the LDM^T factorization if A is symmetric.

Theorem 4.1.2 *If $A = LDM^T$ is the LDM^T factorization of a nonsingular symmetric matrix A, then $L = M$.*

Proof. The matrix $M^{-1}AM^{-T} = M^{-1}LD$ is both symmetric and lower triangular and therefore diagonal. Since D is nonsingular, this implies that $M^{-1}L$ is also diagonal. But $M^{-1}L$ is unit lower triangular and so $M^{-1}L = I$.□

In view of this result, it is possible to halve the work in Algorithm 4.1.1 when it is applied to a symmetric matrix. In the jth step we already know $M(j, 1{:}j-1)$ since $M = L$ and we presume knowledge of L's first $j-1$ columns. Recall that in the jth step of (4.1.2) the vector $v(1{:}j)$ is defined by the first j components of DM^Te_j. Since $M = L$, this says that

$$v(1{:}j) = \begin{bmatrix} d(1)L(j,1) \\ \vdots \\ d(j-1)L(j,j-1) \\ d(j) \end{bmatrix}.$$

Hence, the vector $v(1{:}j-1)$ can be obtained by a simple scaling of L's jth row. The formula $v(j) = A(j,j) - L(j,1{:}j-1)v(1{:}j-1)$ can be derived from the jth equation in $L(1{:}j,1{:}j)v = A(1{:}j,j)$ rendering

```
for j = 1:n
    for i = 1:j-1
        v(i) = L(j,i)d(i)
    end
    v(j) = A(j,j) - L(j,1:j-1)v(1:j-1)
    d(j) = v(j)
    L(j+1:n,j) =
        (A(j+1:n,j) - L(j+1:n,1:j-1)v(1:j-1))/v(j)
end
```

With overwriting this becomes

Algorithm 4.1.2 (LDL^T) If $A \in \mathbb{R}^{n \times n}$ is symmetric and has an LU factorization then this algorithm computes a unit lower triangular matrix L and a diagonal matrix $D = \text{diag}(d_1, \ldots, d_n)$ so $A = LDL^T$. The entry a_{ij} is overwritten with ℓ_{ij} if $i > j$ and with d_i if $i = j$.

```
for j = 1:n
    { Compute v(1:j). }
    for i = 1:j-1
        v(i) = A(j,i)A(i,i)
    end
    v(j) = A(j,j) - A(j,1:j-1)v(1:j-1)
    { Store d(j) and compute L(j+1:n,j). }
    A(j,j) = v(j)
    A(j+1:n,j) =
        (A(j+1:n,j) - A(j+1:n,1:j-1)v(1:j-1))/v(j)
end
```

This algorithm requires $n^3/3$ flops, about half the number of flops involved in Gaussian elimination.

In the next section, we show that if A is both symmetric and positive definite, then Algorithm 4.1.2 not only runs to completion, but is extremely stable. If A is symmetric but not positive definite, then pivoting may be necessary and the methods of §4.4 are relevant.

Example 4.1.2

$$A = \begin{bmatrix} 10 & 20 & 30 \\ 20 & 45 & 80 \\ 30 & 80 & 171 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 4 & 1 \end{bmatrix} \begin{bmatrix} 10 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 4 \\ 0 & 0 & 1 \end{bmatrix}$$

and so if Algorithm 4.1.2 is applied, A is overwritten by

$$A = \begin{bmatrix} 10 & 20 & 30 \\ 2 & 5 & 80 \\ 3 & 4 & 1 \end{bmatrix}.$$

Problems

P4.1.1 Show that the LDM^T factorization of a nonsingular A is unique if it exists.

P4.1.2 Modify Algorithm 4.1.1 so that it computes a factorization of the form $PA = LDM^T$, where L and M are both unit lower triangular, D is diagonal, and P is a permutation that is chosen so $|\ell_{ij}| \le 1$.

P4.1.3 Suppose the n-by-n symmetric matrix $A = (a_{ij})$ is stored in a vector c as follows: $c = (a_{11}, a_{21}, \ldots, a_{n1}, a_{22}, \ldots, a_{n2}, \ldots, a_{nn})$. Rewrite Algorithm 4.1.2 with A stored in this fashion. Get as much indexing outside the inner loops as possible.

P4.1.4 Rewrite Algorithm 4.1.2 for A stored by diagonal. See §1.2.8.

Notes and References for Sec. 4.1

Algorithm 4.1.1 is related to the methods of Crout and Doolittle in that outer product updates are avoided. See Chapter 4 of Fox (1964) or Stewart (1973,131–149). An Algol procedure may be found in

H.J. Bowdler, R.S. Martin, G. Peters, and J.H. Wilkinson (1966), "Solution of Real and Complex Systems of Linear Equations," *Numer. Math. 8*, 217–234.

See also

G.E. Forsythe (1960). "Crout with Pivoting," *Comm. ACM 3*, 507–08.
W.M. McKeeman (1962). "Crout with Equilibration and Iteration," *Comm. ACM 5*, 553–55.

Just as algorithms can be tailored to exploit structure, so can error analysis and perturbation theory:

M. Arioli, J. Demmel, and I. Duff (1989). "Solving Sparse Linear Systems with Sparse Backward Error," *SIAM J. Matrix Anal. Appl. 10*, 165–190.
J.R. Bunch, J.W. Demmel, and C.F. Van Loan (1989). "The Strong Stability of Algorithms for Solving Symmetric Linear Systems," *SIAM J. Matrix Anal. Appl. 10*, 494–499.
A. Barrlund (1991). "Perturbation Bounds for the LDL^T and LU Decompositions," *BIT 31*, 358–363.
D.J. Higham and N.J. Higham (1992). "Backward Error and Condition of Structured Linear Systems," *SIAM J. Matrix Anal. Appl. 13*, 162–175.

4.2 Positive Definite Systems

A matrix $A \in \mathbb{R}^{n \times n}$ is *positive definite* if $x^T Ax > 0$ for all nonzero $x \in \mathbb{R}^n$. Positive definite systems constitute one of the most important classes of special $Ax = b$ problems. Consider the 2-by-2 symmetric case. If

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

is positive definite then

$$\begin{aligned} x &= (1,\ 0)^T &\Rightarrow\ x^TAx &= a_{11} > 0 \\ x &= (0,1)^T &\Rightarrow\ x^TAx &= a_{22} > 0 \\ x &= (1,\ 1)^T &\Rightarrow\ x^TAx &= a_{11} + 2a_{12} + a_{22} > 0 \\ x &= (1,\ -1)^T &\Rightarrow\ x^TAx &= a_{11} - 2a_{12} + a_{22} > 0\,. \end{aligned}$$

The last two equations imply $|a_{12}| \le (a_{11} + a_{22})/2$. From these results we see that the largest entry in A is on the diagonal and that it is positive. This turns out to be true in general. A symmetric positive definite matrix has a "weighty" diagonal. The mass on the diagonal is not blatantly obvious

as in the case of diagonal dominance but it has the same effect in that it precludes the need for pivoting. See §3.4.10.

We begin with a few comments about the property of positive definiteness and what it implies in the unsymmetric case with respect to pivoting. We then focus on the efficient organization of the Cholesky procedure which can be used to safely factor a symmetric positive definite A. Gaxpy, outer product, and block versions are developed. The section concludes with a few comments about the semidefinite case.

4.2.1 Positive Definiteness

Suppose $A \in \mathbb{R}^{n\times n}$ is positive definite. It is obvious that a positive definite matrix is nonsingular for otherwise we could find a nonzero x so $x^TAx = 0$. However, much more is implied by the positivity of the *quadratic form* x^TAx as the following results show.

Theorem 4.2.1 *If $A \in \mathbb{R}^{n\times n}$ is positive definite and $X \in \mathbb{R}^{n\times k}$ has rank k, then $B = X^TAX \in \mathbb{R}^{k\times k}$ is also positive definite.*

Proof. If $z \in \mathbb{R}^k$ satisfies $0 \geq z^TBz = (Xz)^TA(Xz)$ then $Xz = 0$. But since X has full column rank, this implies that $z = 0$. □

Corollary 4.2.2 *If A is positive definite then all its principal submatrices are positive definite. In particular, all the diagonal entries are positive.*

Proof. If $v \in \mathbb{R}^k$ is an integer vector with $1 \leq v_1 < \cdots < v_k \leq n$, then $X = I_n(:, v)$ is a rank k matrix made up columns $v_1, \ldots, v_k$ of the identity. It follows from Theorem 4.2.1 that $A(v, v) = X^TAX$ is positive definite. □

Corollary 4.2.3 *If A is positive definite then the factorization $A = LDM^T$ exists and $D = \text{diag}(d_1, \ldots, d_n)$ has positive diagonal entries.*

Proof. From Corollary 4.2.2, it follows that the submatrices $A(1{:}k, 1{:}k)$ are nonsingular for $k = 1{:}n$ and so from Theorem 4.1.1 the factorization $A = LDM^T$ exists. If we apply Theorem 4.2.1 with $X = L^{-T}$ then $B = DM^TL^{-T} = L^{-1}AL^{-T}$ is positive definite. Since M^TL^{-T} is unit upper triangular, B and D have the same diagonal and it must be positive. □

There are several typical situations that give rise to positive definite matrices in practice:

- The quadratic form is an energy function whose positivity is guaranteed from physical principles.
- The matrix A equals a *cross-product* X^TX where X has full column rank. (Positive definiteness follows by setting $A = I_n$ in Theorem 4.2.1.)
- Both A and A^T are diagonally dominant and each a_{ii} is positive.

4.2.2 Unsymmetric Positive Definite Systems

The mere existence of an LDM^T factorization does not mean that its computation is advisable because the resulting factors may have unacceptably large elements. For example, if $\epsilon > 0$ then the matrix

$$A = \begin{bmatrix} \epsilon & m \\ -m & \epsilon \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -m/\epsilon & 1 \end{bmatrix} \begin{bmatrix} \epsilon & 0 \\ 0 & \epsilon + m^2/\epsilon \end{bmatrix} \begin{bmatrix} 1 & m/\epsilon \\ 0 & 1 \end{bmatrix}$$

is positive definite. But if $m/\epsilon \gg 1$, then pivoting is recommended.

The following result suggests when to expect element growth in the LDM^T factorization of a positive definite matrix.

Theorem 4.2.4 *Let $A \in \mathbb{R}^{n\times n}$ be positive definite and set $T = (A + A^T)/2$ and $S = (A - A^T)/2$. If $A = LDM^T$, then*

$$\| \, |L||D||M^T| \, \|_F \;\leq\; n \left(\| T \|_2 + \| ST^{-1}S \|_2 \right) \tag{4.2.1}$$

Proof. See Golub and Van Loan (1979). □

The theorem suggests when it is safe not to pivot. Assume that the computed factors $\hat{L}$, $\hat{D}$, and $\hat{M}$ satisfy:

$$\| \, |\hat{L}||\hat{D}||\hat{M}^T| \, \|_F \;\leq\; c \| \, |L||D||M^T| \, \|_F, \tag{4.2.2}$$

where c is a constant of modest size. It follows from (4.2.1) and the analysis in §3.3 that if these factors are used to compute a solution to $Ax = b$, then the computed solution $\hat{x}$ satisfies $(A + E)\hat{x} = b$ with

$$\| E \|_F \;\leq\; \mathbf{u} \left(3n \| A \|_F + 5cn^2 \left(\| T \|_2 + \| ST^{-1}S \|_2 \right) \right) + O(\mathbf{u}^2). \tag{4.2.3}$$

It is easy to show that $\| T \|_2 \leq \| A \|_2$, and so it follows that if

$$\Omega \;=\; \frac{\| ST^{-1}S \|_2}{\| A \|_2} \tag{4.2.4}$$

is not too large then it is safe not to pivot. In other words, the norm of the skew part S has to be modest relative to the condition of the symmetric part T. Sometimes it is possible to estimate Ω in an application. This is trivially the case when A is symmetric for then $\Omega = 0$.

4.2.3 Symmetric Positive Definite Systems

When we apply the above results to a symmetric positive definite system we know that the factorization $A = LDL^T$ exists and moreover is stable to compute. However, in this situation another factorization is available.

Theorem 4.2.5 (Cholesky Factorization) *If $A \in \mathbb{R}^{n\times n}$ is symmetric positive definite, then there exists a unique lower triangular $G \in \mathbb{R}^{n\times n}$ with positive diagonal entries such that $A = GG^T$.*

Proof. From Theorem 4.1.2, there exists a unit lower triangular L and a diagonal $D = \text{diag}(d_1, \ldots, d_n)$ such that $A = LDL^T$. Since the d_k are positive, the matrix $G = L\,\text{diag}(\sqrt{d_1}, \ldots, \sqrt{d_n})$ is real lower triangular with positive diagonal entries. It also satisfies $A = GG^T$. Uniqueness follows from the uniqueness of the LDL^T factorization. □

The factorization $A = GG^T$ is known as the *Cholesky factorization* and G is referred to as the *Cholesky triangle*. Note that if we compute the Cholesky factorization and solve the triangular systems $Gy = b$ and $G^Tx = y$, then $b = Gy = G(G^Tx) = (GG^T)x = Ax$.

Our proof of the Cholesky factorization in Theorem 4.2.5 is constructive. However, more effective methods for computing the Cholesky triangle can be derived by manipulating the equation $A = GG^T$. This can be done in several ways as we show in the next few subsections.

Example 4.2.1 The matrix

$$\begin{bmatrix} 2 & -2 \\ -2 & 5 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}\begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}\begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \sqrt{2} & 0 \\ -\sqrt{2} & \sqrt{3} \end{bmatrix}\begin{bmatrix} \sqrt{2} & -\sqrt{2} \\ 0 & \sqrt{3} \end{bmatrix}$$

is positive definite.

4.2.4 Gaxpy Cholesky

We first derive an implementation of Cholesky that is rich in the gaxpy operation. If we compare jth columns in the equation $A = GG^T$ then we obtain

$$A(:,j) = \sum_{k=1}^{j} G(j,k)G(:,k).$$

This says that

$$G(j,j)G(:,j) \;=\; A(:,j) - \sum_{k=1}^{j-1} G(j,k)G(:,k) \;\equiv\; v. \qquad (4.2.5)$$

If we know the first $j-1$ columns of G, then v is computable. It follows by equating components in (4.2.5) that

$$G(j{:}n,j) = v(j{:}n)/\sqrt{v(j)}.$$

This is a scaled gaxpy operation and so we obtain the following gaxpy-based method for computing the Cholesky factorization:


```
for j = 1:n
    v(j:n) = A(j:n, j)
    for k = 1:j - 1
        v(j:n) = v(j:n) - G(j,k)G(j:n,k)
    end
    G(j:n, j) = v(j:n)/sqrt(v(j))
end
```

It is possible to arrange the computations so that G overwrites the lower triangle of A.

Algorithm 4.2.1 (Cholesky: Gaxpy Version) Given a symmetric positive definite $A \in \mathbb{R}^{n\times n}$, the following algorithm computes a lower triangular $G \in \mathbb{R}^{n\times n}$ such that $A = GG^T$. For all $i \geq j$, $G(i,j)$ overwrites $A(i,j)$.

```
for j = 1:n
    if j > 1
        A(j:n, j) = A(j:n, j) - A(j:n, 1:j - 1)A(j, 1:j - 1)^T
    end
    A(j:n, j) = A(j:n, j)/sqrt(A(j,j))
end
```

This algorithm requires $n^3/3$ flops.

4.2.5 Outer Product Cholesky

An alternative Cholesky procedure based on outer product (rank-1) updates can be derived from the partitioning

$$A = \begin{bmatrix} \alpha & v^T \\ v & B \end{bmatrix} = \begin{bmatrix} \beta & 0 \\ v/\beta & I_{n-1} \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 0 & B - vv^T/\alpha \end{bmatrix}\begin{bmatrix} \beta & v^T/\beta \\ 0 & I_{n-1} \end{bmatrix}. \qquad (4.2.6)$$

Here, $\beta = \sqrt{\alpha}$ and we know that $\alpha > 0$ because A is positive definite. Note that $B - vv^T/\alpha$ is positive definite because it is a principal submatrix of X^TAX where

$$X = \begin{bmatrix} 1 & -v^T/\alpha \\ 0 & I_{n-1} \end{bmatrix}.$$

If we have the Cholesky factorization $G_1G_1^T = B - vv^T/\alpha$, then from (4.2.6) it follows that $A = GG^T$ with

$$G = \begin{bmatrix} \beta & 0 \\ v/\beta & G_1 \end{bmatrix}.$$

Thus, the Cholesky factorization can be obtained through the repeated application of (4.2.6), much in the the style of kji Gaussian elimination.

Algorithm 4.2.2 (Cholesky: Outer product Version) Given a symmetric positive definite $A \in \mathbb{R}^{n\times n}$, the following algorithm computes a lower triangular $G \in \mathbb{R}^{n\times n}$ such that $A = GG^T$. For all $i \geq j$, $G(i,j)$ overwrites $A(i,j)$.

```
for k = 1:n
    A(k,k) = sqrt(A(k,k))
    A(k+1:n,k) = A(k+1:n,k)/A(k,k)
    for j = k+1:n
        A(j:n,j) = A(j:n,j) - A(j:n,k)A(j,k)
    end
end
```

This algorithm involves $n^3/3$ flops. Note that the j-loop computes the lower triangular part of the outer product update

$$A(k+1:n, k+1:n) \;=\; A(k+1:n, k+1:n) \;-\; A(k+1:n,k)A(k+1:n,k)^T.$$

Recalling our discussion in §1.4.8 about gaxpy versus outer product updates, it is easy to show that Algorithm 4.2.1 involves fewer vector touches than Algorithm 4.2.2 by a factor of two.

4.2.6 Block Dot Product Cholesky

Suppose $A \in \mathbb{R}^{n\times n}$ is symmetric positive definite. Regard $A = (A_{ij})$ and its Cholesky factor $G = (G_{ij})$ as N-by-N block matrices with square diagonal blocks. By equating (i,j) blocks in the equation $A = GG^T$ with $i \geq j$ it follows that

$$A_{ij} \;=\; \sum_{k=1}^{j} G_{ik}G_{jk}^T.$$

Defining

$$S \;=\; A_{ij} - \sum_{k=1}^{j-1} G_{ik}G_{jk}^T$$

we see that $G_{jj}G_{jj}^T = S$ if $i = j$ and that $G_{ij}G_{jj}^T = S$ if $i > j$. Properly sequenced, these equations can be arranged to compute all the G_{ij}:

Algorithm 4.2.3 (Cholesky: Block Dot Product Version) Given a symmetric positive definite $A \in \mathbb{R}^{n\times n}$, the following algorithm computes a lower triangular $G \in \mathbb{R}^{n\times n}$ such that $A = GG^T$. The lower triangular part of A is overwritten by the lower triangular part of G. A is regarded as an N-by-N block matrix with square diagonal blocks.


```
for j = 1:N
    for i = j:N
        S = A_ij - sum_{k=1}^{j-1} G_ik G_jk^T
        if i = j
            Compute Cholesky factorization S = G_jj G_jj^T.
        else
            Solve G_ij G_jj^T = S for G_ij
        end
        Overwrite A_ij with G_ij.
    end
end
```

The overall process involves $n^3/3$ flops like the other Cholesky procedures that we have developed. The procedure is rich in matrix multiplication assuming a suitable blocking of the matrix A. For example, if $n = rN$ and each A_{ij} is r-by-r, then the level-3 fraction is approximately $1 - (1/N^2)$.

Algorithm 4.2.3 is incomplete in the sense that we have not specified how the products $G_{ik}G_{jk}$ are formed or how the r-by-r Cholesky factorizations $S = G_{jj}G_{jj}^T$ are computed. These important details would have to be worked out carefully in order to extract high performance.

Another block procedure can be derived from the gaxpy Cholesky algorithm. After r steps of Algorithm 4.2.1 we know the matrices $G_{11} \in \mathbb{R}^{r\times r}$ and $G_{21} \in \mathbb{R}^{(n-r)\times r}$ in

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} G_{11} & 0 \\ G_{21} & I_{n-r} \end{bmatrix} \begin{bmatrix} I_r & 0 \\ 0 & \tilde{A} \end{bmatrix} \begin{bmatrix} G_{11} & 0 \\ G_{21} & I_{n-r} \end{bmatrix}^T.$$

We then perform r more steps of gaxpy Cholesky not on A but on the reduced matrix $\tilde{A} = A_{22} - G_{21}G_{21}^T$ which we *explicitly* form exploiting symmetry. Continuing in this way we obtain a block Cholesky algorithm whose kth step involves r gaxpy Cholesky steps on a matrix of order $n - (k-1)r$ followed a level-3 computation having order $n - kr$. The level-3 fraction is approximately equal to $1 - 3/(2N)$ if $n \approx rN$.

4.2.7 Stability of the Cholesky Process

In exact arithmetic, we know that a symmetric positive definite matrix has a Cholesky factorization. Conversely, if the Cholesky process runs to completion with strictly positive square roots, then A is positive definite. Thus, to find out if a matrix A is positive definite, we merely try to compute its Cholesky factorization using any of the methods given above.

The situation in the context of roundoff error is more interesting. The numerical stability of the Cholesky algorithm roughly follows from the in-

equality

$$g_{ij}^2 \le \sum_{k=1}^{i} g_{ik}^2 = a_{ii}.$$

This shows that the entries in the Cholesky triangle are nicely bounded. The same conclusion can be reached from the equation $\| G \|_2^2 = \| A \|_2$.

The roundoff errors associated with the Cholesky factorization have been extensively studied in a classical paper by Wilkinson (1968). Using the results in this paper, it can be shown that if $\hat{x}$ is the computed solution to $Ax = b$, obtained via any of our Cholesky procedures then $\hat{x}$ solves the perturbed system $(A + E)\hat{x} = b$ where $\| E \|_2 \le c_n \mathbf{u} \| A \|_2$ and c_n is a small constant depending upon n. Moreover, Wilkinson shows that if $q_n \mathbf{u} \kappa_2(A) \le 1$ where q_n is another small constant, then the Cholesky process runs to completion, i.e, no square roots of negative numbers arise.

Example 4.2.2 If Algorithm 4.2.2 is applied to the positive definite matrix

$$A = \begin{bmatrix} 100 & 15 & .01 \\ 15 & 2.3 & .01 \\ .01 & .01 & 1.00 \end{bmatrix}$$

and $\beta = 10$, $t = 2$, rounded arithmetic used, then $\hat{g}_{11} = 10$, $\hat{g}_{21} = 1.5$, $\hat{g}_{31} = .001$ and $\hat{g}_{22} = 0.00$. The algorithm then breaks down trying to compute g_{32}.

4.2.8 The Semidefinite Case

A matrix is said to be *positive semidefinite* if $x^T Ax \ge 0$ for all vectors x. Symmetric positive semidefinite (*sps*) matrices are important and we briefly discuss some Cholesky-like manipulations that can be used to solve various *sps* problems. Results about the diagonal entries in an *sps* matrix are needed first.

Theorem 4.2.6 *If $A \in \mathbb{R}^{n \times n}$ is symmetric positive semidefinite, then*

$$|a_{ij}| \le (a_{ii} + a_{jj})/2 \qquad (4.2.7)$$

$$|a_{ij}| \le \sqrt{a_{ii}a_{jj}} \quad (i \ne j) \qquad (4.2.8)$$

$$\max_{i,j} |a_{ij}| = \max_{i} a_{ii} \qquad (4.2.9)$$

$$a_{ii} = 0 \ \Rightarrow\ A(i,:) = 0,\ A(:,i) = 0 \qquad (4.2.10)$$

Proof. If $x = e_i + e_j$ then $0 \le x^T Ax = a_{ii} + a_{jj} + 2a_{ij}$ while $x = e_i - e_j$ implies $0 \le x^T Ax = a_{ii} + a_{jj} - 2a_{ij}$. Inequality (4.2.7) follows from these two results. Equation (4.2.9) is an easy consequence of (4.2.7).

To prove (4.2.8) assume without loss of generality that $i = 1$ and $j = 2$ and consider the inequality

$$0 \le \begin{bmatrix} x \\ 1 \end{bmatrix}^T \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x \\ 1 \end{bmatrix} = a_{11}x^2 + 2a_{12}x + a_{22}$$

which holds since $A(1{:}2, 1{:}2)$ is also semidefinite. This is a quadratic equation in x and for the inequality to hold, the discriminant $4a_{12}^2 - 4a_{11}a_{22}$ must be negative. Implication (4.2.10) follows from (4.2.8). □

Consider what happens when outer product Cholesky is applied to an *sps* matrix. If a zero $A(k,k)$ is encountered then from (4.2.10) $A(k{:}n, k)$ is zero and there is "nothing to do" and we obtain

```
for k = 1:n
    if A(k,k) > 0
        A(k,k) = sqrt(A(k,k))
        A(k+1:n,k) = A(k+1:n,k)/A(k,k)
        for j = k+1:n
            A(j:n,j) = A(j:n,j) - A(j:n,k)A(j,k)        (4.2.11)
        end
    end
end
```

Thus, a simple change makes Algorithm 4.2.2 applicable to the semidefinite case. However, in practice rounding errors preclude the generation of exact zeros and it may be preferable to incorporate pivoting.

4.2.9 Symmetric Pivoting

To preserve symmetry in a symmetric A we only consider data reorderings of the form PAP^T where P is a permutation. Row permutations ($A \leftarrow PA$) or column permutations ($A \leftarrow AP$) alone destroy symmetry. An update of the form

$$A \leftarrow PAP^T$$

is called a *symmetric permutation* of A. Note that such an operation *does not* move off-diagonal elements to the diagonal. The diagonal of PAP^T is a reordering of the diagonal of A.

Suppose at the beginning of the kth step in (4.2.11) we symmetrically permute the largest diagonal entry of $A(k{:}n, k{:}n)$ into the lead position. If that largest diagonal entry is zero then $A(k{:}n, k{:}n) = 0$ by virtue of (4.2.10). In this way we can compute the factorization $PAP^T = GG^T$ where $G \in \mathbb{R}^{n \times (k-1)}$ is lower triangular.

Algorithm 4.2.4 Suppose $A \in \mathbb{R}^{n \times n}$ is symmetric positive semidefinite and that $\text{rank}(A) = r$. The following algorithm computes a permutation P, the index r, and an n-by-r lower triangular matrix G such that $PAP^T = GG^T$. The lower triangular part of $A(:, 1{:}r)$ is overwritten by the lower triangular part of G. $P = P_r \cdots P_1$ where P_k is the identity with rows k and $piv(k)$ interchanged.

$r = 0$
for $k = 1{:}n$
  Find q $(k \le q \le n)$ so $A(q,q) = \max\{A(k,k), .., A(n,n)\}$
  if $A(q,q) > 0$
    $r = r + 1$
    $piv(k) = q$
    $A(k,:) \leftrightarrow A(q,:)$
    $A(:,k) \leftrightarrow A(:,q)$
    $A(k,k) = \sqrt{A(k,k)}$
    $A(k+1{:}n, k) = A(k+1{:}n, k)/A(k,k)$
    for $j = k+1{:}n$
      $A(j{:}n, j) = A(j{:}n, j) - A(j{:}n, k)A(j,k)$
    end
  end
end

In practice, a tolerance is used to detect small $A(k,k)$. However, the situation is quite tricky and the reader should consult Higham (1989). In addition, §5.5 has a discussion of tolerances in the rank detection problem. Finally, we remark that a truly efficient implementation of Algorithm 4.2.4 would only access the lower triangular portion of A.

4.2.10 The Polar Decomposition and Square Root

Let $A = U_1\Sigma_1V^T$ be the thin SVD of $A \in \mathbb{R}^{m\times n}$ where $m \ge n$. Note that

$$A = (U_1V^T)(V\Sigma_1V^T) \equiv ZP \qquad (4.2.12)$$

where $Z = U_1V^T$ and $P = V\Sigma_1V^T$. Z has orthonormal columns and P is symmetric positive semidefinite because

$$x^TPx = (V^Tx)^T\Sigma_1(V^Tx) = \sum_{k=1}^{n}\sigma_k y_k^2 \ge 0$$

where $y = V^Tx$. The decomposition (4.2.12) is called the *polar decomposition* because it is analogous to the complex number factorization $z = e^{i \cdot arg(z)}|z|$. See §12.4.1 for further discussion.

Another important decomposition is the *matrix square root*. Suppose $A \in \mathbb{R}^{n\times n}$ is symmetric positive semidefinite and that $A = GG^T$ is its Cholesky factorization. If $G = U\Sigma V^T$ is G's SVD and $X = U\Sigma U^T$, then X is symmetric positive semidefinite and

$$A = GG^T = (U\Sigma V^T)(U\Sigma V^T)^T = U\Sigma^2U^T = (U\Sigma U^T)(U\Sigma U^T) = X^2.$$

Thus, X is a *square root* of A. It can be shown (most easily with eigenvalue theory) that a symmetric positive semidefinite matrix has a unique symmetric positive semidefinite square root.

Problems

P4.2.1 Suppose that $H = A + iB$ is Hermitian and positive definite with $A, B \in \mathbb{R}^{n\times n}$. This means that $x^HHx > 0$ whenever $x \ne 0$. (a) Show that

$$C = \begin{bmatrix} A & -B \\ B & A \end{bmatrix}$$

is symmetric and positive definite. (b) Formulate an algorithm for solving $(A+iB)(x+iy) = (b+ic)$, where b, c, x, and y are in $\mathbb{R}^n$. It should involve $8n^3/3$ flops. How much storage is required?

P4.2.2 Suppose $A \in \mathbb{R}^{n\times n}$ is symmetric and positive definite. Give an algorithm for computing an upper triangular matrix $R \in \mathbb{R}^{n\times n}$ such that $A = RR^T$.

P4.2.3 Let $A \in \mathbb{R}^{n\times n}$ be positive definite and set $T = (A+A^T)/2$ and $S = (A-A^T)/2$. (a) Show that $\| A^{-1} \|_2 \le \| T^{-1} \|_2$ and $x^TA^{-1}x \le x^TT^{-1}x$ for all $x \in \mathbb{R}^n$. (b) Show that if $A = LDM^T$, then $d_k \ge 1/\| T^{-1} \|_2$ for $k = 1{:}n$

P4.2.4 Find a 2-by-2 real matrix A with the property that $x^TAx > 0$ for all real nonzero 2-vectors but which is not positive definite when regarded as a member of $\mathbb{C}^{2\times 2}$.

P4.2.5 Suppose $A \in \mathbb{R}^{n\times n}$ has a positive diagonal. Show that if both A and A^T are strictly diagonally dominant, then A is positive definite.

P4.2.6 Show that the function $f(x) = (x^TAx)/2$ is a vector norm on $\mathbb{R}^n$ if and only if A is positive definite.

P4.2.7 Modify Algorithm 4.2.1 so that if the square root of a negative number is encountered, then the algorithm finds a unit vector x so $x^TAx < 0$ and terminates.

P4.2.8 The numerical range $W(A)$ of a complex matrix A is defined to be the set $W(A) = \{x^HAx : x^Hx = 1\}$. Show that if $0 \notin W(A)$, then A has an LU factorization.

P4.2.9 Formulate an $m < n$ version of the polar decomposition for $A \in \mathbb{R}^{m\times n}$.

P4.2.10 Suppose $A = I + uu^T$ where $A \in \mathbb{R}^{n\times n}$ and $\| u \|_2 = 1$. Give explicit formulae for the diagonal and subdiagonal of A's Cholesky factor.

P4.2.11 Suppose $A \in \mathbb{R}^{n\times n}$ is symmetric positive definite and that its Cholesky factor is available. Let $e_k = I_n(:,k)$. For $1 \le i < j \le n$, let α_{ij} be the smallest real that makes $A + \alpha(e_ie_j^T + e_je_i^T)$ singular. Likewise, let α_{ii} be the smallest real that makes $(A + \alpha e_ie_i^T)$ singular. Show how to compute these quantities using the Sherman-Morrison-Woodbury formula. How many flops are required to find all the α_{ij}?

Notes and References for Sec. 4.2

The definiteness of the quadratic form x^TAx can frequently be established by considering the mathematics of the underlying problem. For example, the discretization of certain partial differential operators gives rise to provably positive definite matrices. Aspects of the unsymmetric positive definite problem are discussed in

A. Buckley (1974). "A Note on Matrices $A = I + H$, H Skew-Symmetric," *Z. Angew. Math. Mech.* *54*, 125–26.

A. Buckley (1977). "On the Solution of Certain Skew-Symmetric Linear Systems," *SIAM J. Num. Anal.* *14*, 566–70.

G.H. Golub and C. Van Loan (1979). "Unsymmetric Positive Definite Linear Systems," *Lin. Alg. and Its Applic.* *28*, 85–98.

R. Mathias (1992). "Matrices with Positive Definite Hermitian Part: Inequalities and Linear Systems," *SIAM J. Matrix Anal. Appl.* *13*, 640–654.

Symmetric positive definite systems constitute the most important class of special $Ax = b$ problems. Algol programs for these problems are given in

R.S. Martin, G. Peters, and J.H. Wilkinson (1965). "Symmetric Decomposition of a Positive Definite Matrix," *Numer. Math.* *7*, 362-83.

R.S. Martin, G. Peters, and J.H. Wilkinson (1966). "Iterative Refinement of the Solution of a Positive Definite System of Equations," *Numer. Math.* *8*, 203–16.

F.L. Bauer and C. Reinsch (1971). "Inversion of Positive Definite Matrices by the Gauss-Jordan Method," in *Handbook for Automatic Computation Vol. 2, Linear Algebra*, J.H. Wilkinson and C. Reinsch, eds. Springer-Verlag, New York, 45–49.

The roundoff errors associated with the method are analyzed in

J.H. Wilkinson (1968). "À Priori Error Analysis of Algebraic Processes," *Proc. International Congress Math.* (Moscow: Izdat. Mir, 1968), pp. 629–39.

J. Meinguet (1983). "Refined Error Analyses of Cholesky Factorization," *SIAM J. Numer. Anal.* *20*, 1243–1250.

A. Kielbasinski (1987). "A Note on Rounding Error Analysis of Cholesky Factorization," *Lin. Alg. and Its Applic.* *88/89*, 487–494.

N.J. Higham (1990). "Analysis of the Cholesky Decomposition of a Semidefinite Matrix," in *Reliable Numerical Computation*, M.G. Cox and S.J. Hammarling (eds), Oxford University Press, Oxford, UK, 161–185.

R. Carter (1991). "Y-MP Floating Point and Cholesky Factorization," *Int'l J. High Speed Computing* *3*, 215–222.

J-Guang Sun (1992). "Rounding Error and Perturbation Bounds for the Cholesky and LDL^T Factorizations," *Lin. Alg. and Its Applic.* *173*, 77–97.

The question of how the Cholesky triangle G changes when $A = GG^T$ is perturbed is analyzed in

G.W. Stewart (1977b). "Perturbation Bounds for the QR Factorization of a Matrix," *SIAM J. Num. Anal.* *14*, 509-18.

Z. Dramăc, M. Omladič, and K. Veselič (1994). "On the Perturbation of the Cholesky Factorization," *SIAM J. Matrix Anal. Appl.* *15*,1319–1332.

Nearness/sensitivity issues associated with positive semi-definiteness and the polar decomposition are presented in

N.J. Higham (1988). "Computing a Nearest Symmetric Positive Semidefinite Matrix," *Lin. Alg. and Its Applic.* *103*, 103–118.

R. Mathias (1993). "Perturbation Bounds for the Polar Decomposition," *SIAM J. Matrix Anal. Appl.* *14*, 588–597.

R-C. Li (1995). "New Perturbation Bounds for the Unitary Polar Factor," *SIAM J. Matrix Anal. Appl.* *16*, 327–332.

Computationally-oriented references for the polar decomposition and the square root are given in §8.6 and §11.2 respectively.

4.3 Banded Systems

In many applications that involve linear systems, the matrix of coefficients is *banded*. This is the case whenever the equations can be ordered so that each unknown x_i appears in only a few equations in a "neighborhood" of the ith equation. Formally, we say that $A = (a_{ij})$ has *upper bandwidth* q if $a_{ij} = 0$ whenever $j > i + q$ and *lower bandwidth* p if $a_{ij} = 0$ whenever $i > j + p$. Substantial economies can be realized when solving banded systems because the triangular factors in LU, GG^T, LDM^T, etc., are also banded.

Before proceeding the reader is advised to review §1.2 where several aspects of band matrix manipulation are discussed.

4.3.1 Band LU Factorization

Our first result shows that if A is banded and $A = LU$ then $L(U)$ inherits the lower (upper) bandwidth of A.

Theorem 4.3.1 *Suppose $A \in \mathbb{R}^{n\times n}$ has an LU factorization $A = LU$. If A has upper bandwidth q and lower bandwidth p, then U has upper bandwidth q and L has lower bandwidth p.*

Proof. The proof is by induction on n. From (3.2.6) we have the factorization

$$A = \begin{bmatrix} \alpha & w^T \\ v & B \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ v/\alpha & I_{n-1} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & B - vw^T/\alpha \end{bmatrix} \begin{bmatrix} \alpha & w^T \\ 0 & I_{n-1} \end{bmatrix}.$$

It is clear that $B - vw^T/\alpha$ has upper bandwidth q and lower bandwidth p because only the first q components of w and the first p components of v are nonzero. Let L_1U_1 be the LU factorization of this matrix. Using the induction hypothesis and the sparsity of w and v, it follows that

$$L = \begin{bmatrix} 1 & 0 \\ v/\alpha & L_1 \end{bmatrix} \quad \text{and} \quad U = \begin{bmatrix} \alpha & w^T \\ 0 & U_1 \end{bmatrix}$$

have the desired bandwidth properties and satisfy $A = LU$. □

The specialization of Gaussian elimination to banded matrices having an LU factorization is straightforward.

Algorithm 4.3.1 (Band Gaussian Elimination: Outer Product Version) Given $A \in \mathbb{R}^{n\times n}$ with upper bandwidth q and lower bandwidth p, the following algorithm computes the factorization $A = LU$, assuming it exists. $A(i,j)$ is overwritten by $L(i,j)$ if $i > j$ and by $U(i,j)$ otherwise.

```
for k = 1:n-1
    for i = k+1:min(k+p,n)
        A(i,k) = A(i,k)/A(k,k)
    end
    for j = k+1:min(k+q,n)
        for i = k+1:min(k+p,n)
            A(i,j) = A(i,j) - A(i,k)A(k,j)
        end
    end
end
```

If $n \gg p$ and $n \gg q$ then this algorithm involves about $2npq$ flops. Band versions of Algorithm 4.1.1 (LDM^T) and all the Cholesky procedures also exist, but we leave their formulation to the exercises.

4.3.2 Band Triangular System Solving

Analogous savings can also be made when solving banded triangular systems.

Algorithm 4.3.2 (Band Forward Substitution: Column Version) Let $L \in \mathbb{R}^{n\times n}$ be a unit lower triangular matrix having lower bandwidth p. Given $b \in \mathbb{R}^n$, the following algorithm overwrites b with the solution to $Lx = b$.

```
for j = 1:n
    for i = j+1:min(j+p,n)
        b(i) = b(i) - L(i,j)b(j)
    end
end
```

If $n \gg p$ then this algorithm requires about $2np$ flops.

Algorithm 4.3.3 (Band Back-Substitution: Column Version) Let $U \in \mathbb{R}^{n\times n}$ be a nonsingular upper triangular matrix having upper bandwidth q. Given $b \in \mathbb{R}^n$, the following algorithm overwrites b with the solution to $Ux = b$.

```
for j = n:-1:1
    b(j) = b(j)/U(j,j)
    for i = max(1,j-q):j-1
        b(i) = b(i) - U(i,j)b(j)
    end
end
```

If $n \gg q$ then this algorithm requires about $2nq$ flops.

4.3.3 Band Gaussian Elimination with Pivoting

Gaussian elimination with partial pivoting can also be specialized to exploit band structure in A. If, however, $PA = LU$, then the band properties of L and U are not quite so simple. For example, if A is tridiagonal and the first two rows are interchanged at the very first step of the algorithm, then u_{13} is nonzero. Consequently, row interchanges expand bandwidth. Precisely how the band enlarges is the subject of the following theorem.

Theorem 4.3.2 *Suppose $A \in \mathbb{R}^{n\times n}$ is nonsingular and has upper and lower bandwidths q and p, respectively. If Gaussian elimination with partial pivoting is used to compute Gauss transformations*

$$M_j = I - \alpha^{(j)}e_j^T \qquad j = 1:n-1$$

and permutations $P_1, \ldots, P_{n-1}$ such that $M_{n-1}P_{n-1}\cdots M_1P_1A = U$ is upper triangular, then U has upper bandwidth $p+q$ and $\alpha_i^{(j)} = 0$ whenever $i \le j$ or $i > j+p$.

Proof. Let $PA = LU$ be the factorization computed by Gaussian elimination with partial pivoting and recall that $P = P_{n-1}\cdots P_1$. Write $P^T = [\, e_{s_1}, \ldots, e_{s_n} \,]$, where $\{s_1, ..., s_n\}$ is a permutation of $\{1, 2, ..., n\}$. If $s_i > i+p$ then it follows that the leading i-by-i principal submatrix of PA is singular, since $(PA)_{ij} = a_{s_i,j}$ for $j = 1:s_i - p - 1$ and $s_i - p - 1 \ge i$. This implies that U and A are singular, a contradiction. Thus, $s_i \le i+p$ for $i = 1:n$ and therefore, PA has upper bandwidth $p+q$. It follows from Theorem 4.3.1 that U has upper bandwidth $p+q$.

The assertion about the $\alpha^{(j)}$ can be verified by observing that M_j need only zero elements $(j+1, j), \ldots, (j+p, j)$ of the partially reduced matrix $P_jM_{j-1}P_{j-1}\cdots{}_1P_1A$. □

Thus, pivoting destroys band structure in the sense that U becomes "wider" than A's upper triangle, while nothing at all can be said about the bandwidth of L. However, since the jth column of L is a permutation of the jth Gauss vector α_j, it follows that L has at most $p+1$ nonzero elements per column.

4.3.4 Hessenberg LU

As an example of an unsymmetric band matrix computation, we show how Gaussian elimination with partial pivoting can be applied to factor an upper Hessenberg matrix H. (Recall that if H is upper Hessenberg then $h_{ij} = 0$, $i > j+1$). After $k-1$ steps of Gaussian elimination with partial pivoting

we are left with an upper Hessenberg matrix of the form:

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & \times & \times \end{bmatrix} \qquad k = 3, n = 5$$

By virtue of the special structure of this matrix, we see that the next permutation, P_3, is either the identity or the identity with rows 3 and 4 interchanged. Moreover, the next Gauss transformation M_k has a single nonzero multiplier in the $(k+1, k)$ position. This illustrates the kth step of the following algorithm.

Algorithm 4.3.4 (Hessenberg LU) Given an upper Hessenberg matrix $H \in \mathbb{R}^{n \times n}$, the following algorithm computes the upper triangular matrix $M_{n-1}P_{n-1} \cdots M_1P_1H = U$ where each P_k is a permutation and each M_k is a Gauss transformation whose entries are bounded by unity. $H(i,k)$ is overwritten with $U(i,k)$ if $i \leq k$ and by $(M_k)_{k+1,k}$ if $i = k+1$. An integer vector $piv(1{:}n-1)$ encodes the permutations. If $P_k = I$, then $piv(k) = 0$. If P_k interchanges rows k and $k+1$, then $piv(k) = 1$.

```
for k = 1:n − 1
    if |H(k, k)| < |H(k + 1, k)|
        piv(k) = 1; H(k, k:n) ↔ H(k + 1, k:n)
    else
        piv(k) = 0
    end
    if H(k, k) ≠ 0
        t = −H(k + 1, k)/H(k, k)
        for j = k + 1:n
            H(k + 1, j) = H(k + 1, j) + tH(k, j)
        end
        H(k + 1, k) = t
    end
end
```

This algorithm requires n^2 flops.

4.3.5 Band Cholesky

The rest of this section is devoted to banded $Ax = b$ problems where the matrix A is also symmetric positive definite. The fact that pivoting is unnecessary for such matrices leads to some very compact, elegant algorithms. In particular, it follows from Theorem 4.3.1 that if $A = GG^T$ is the Cholesky factorization of A, then G has the same lower bandwidth as A.

This leads to the following banded version of Algorithm 4.2.1, gaxpy-based Cholesky

Algorithm 4.3.5 (Band Cholesky: Gaxpy Version) Given a symmetric positive definite $A \in \mathbb{R}^{n \times n}$ with bandwidth p, the following algorithm computes a lower triangular matrix G with lower bandwidth p such that $A = GG^T$. For all $i \geq j$, $G(i,j)$ overwrites $A(i,j)$.

```
for j = 1:n
    for k = max(1, j − p):j − 1
        λ = min(k + p, n)
        A(j:λ, j) = A(j:λ, j) − A(j, k)A(j:λ, k)
    end
    λ = min(j + p, n)
    A(j:λ, j) = A(j:λ, j)/√A(j, j)
end
```

If $n \gg p$ then this algorithm requires about $n(p^2 + 3p)$ flops and n square roots. Of course, in a serious implementation an appropriate data structure for A should be used. For example, if we just store the nonzero lower triangular part, then a $(p+1)$-by-n array would suffice. (See §1.2.6)

If our band Cholesky procedure is coupled with appropriate band triangular solve routines then approximately $np^2 + 7np + 2n$ flops and n square roots are required to solve $Ax = b$. For small p it follows that the square roots represent a significant portion of the computation and it is preferable to use the LDL^T approach. Indeed, a careful flop count of the steps $A = LDL^T$, $Ly = b$, $Dz = y$, and $L^Tx = z$ reveals that $np^2 + 8np + n$ flops and no square roots are needed.

4.3.6 Tridiagonal System Solving

As a sample narrow band LDL^T solution procedure, we look at the case of symmetric positive definite tridiagonal systems. Setting

$$L = \begin{bmatrix} 1 & & & \cdots & 0 \\ e_1 & 1 & & & \vdots \\ & \ddots & \ddots & & \\ \vdots & & \ddots & & \\ 0 & \cdots & & e_{n-1} & 1 \end{bmatrix}$$

and $D = \text{diag}(d_1, \ldots, d_n)$ we deduce from the equation $A = LDL^T$ that:

$$\begin{array}{lcll} a_{11} & = & d_1 & \\ a_{k,k-1} & = & e_{k-1}d_{k-1} & k = 2{:}n \\ a_{kk} & = & d_k + e_{k-1}^2 d_{k-1} = d_k + e_{k-1}a_{k,k-1} & k = 2{:}n \end{array}$$

Thus, the d_i and e_i can be resolved as follows:

```
d_1 = a_11
for k = 2:n
    e_{k-1} = a_{k,k-1}/d_{k-1}; d_k = a_kk - e_{k-1} a_{k,k-1}
end
```

To obtain the solution to $Ax = b$ we solve $Ly = b$, $Dz = y$, and $L^T x = z$. With overwriting we obtain

Algorithm 4.3.6 (Symmetric, Tridiagonal, Positive Definite System Solver) Given an n-by-n symmetric, tridiagonal, positive definite matrix A and $b \in \mathbb{R}^n$, the following algorithm overwrites b with the solution to $Ax = b$. It is assumed that the diagonal of A is stored in $d(1{:}n)$ and the superdiagonal in $e(1{:}n-1)$.

```
for k = 2:n
    t = e(k-1); e(k-1) = t/d(k-1); d(k) = d(k) - te(k-1)
end
for k = 2:n
    b(k) = b(k) - e(k-1)b(k-1)
end
b(n) = b(n)/d(n)
for k = n-1: -1:1
    b(k) = b(k)/d(k) - e(k)b(k+1)
end
```

This algorithm requires $8n$ flops.

4.3.7 Vectorization Issues

The tridiagonal example brings up a sore point: narrow band problems and vector/pipeline architectures do not mix well. The narrow band implies short vectors. However, it is sometimes the case that large, independent sets of such problems must be solved at the same time. Let us look at how such a computation should be arranged in light of the issues raised in §1.4.

For simplicity, assume that we must solve the n-by-n unit lower bidiagonal systems

$$A^{(k)}x^{(k)} = b^{(k)} \qquad k = 1{:}m$$

and that $m \gg n$. Suppose we have arrays $E(1{:}n-1, 1{:}m)$ and $B(1{:}n, 1{:}m)$ with the property that $E(1{:}n-1, k)$ houses the subdiagonal of $A^{(k)}$ and $B(1{:}n, k)$ houses the kth right hand side $b^{(k)}$. We can overwrite $b^{(k)}$ with the solution $x^{(k)}$ as follows:

```
for k = 1:m
    for i = 2:n
        B(i,k) = B(i,k) - E(i-1,k)B(i-1,k)
    end
end
```

The problem with this algorithm, which sequentially solves each bidiagonal system in turn, is that the inner loop does not vectorize. This is because of the dependence of $B(i,k)$ on $B(i-1,k)$. If we interchange the k and i loops we get

```
for i = 2:n
    for k = 1:m
        B(i,k) = B(i,k) - E(i-1,k)B(i-1,k)        (4.3.1)
    end
end
```

Now the inner loop vectorizes well as it involves a vector multiply and a vector add. Unfortunately, (4.3.1) is not a unit stride procedure. However, this problem is easily rectified if we store the subdiagonals and right-hand-sides by row. That is, we use the arrays $E(1{:}m, 1{:}n-1)$ and $B(1{:}m, 1{:}n-1)$ and store the subdiagonal of $A^{(k)}$ in $E(k, 1{:}n-1)$ and $b^{(k)^T}$ in $B(k, 1{:}n)$. The computation (4.3.1) then transforms to

```
for i = 2:n
    for k = 1:m
        B(k,i) = B(k,i) - E(k,i-1)B(k,i-1)
    end
end
```

illustrating once again the effect of data structure on performance.

4.3.8 Band Matrix Data Structures

The above algorithms are written as if the matrix A is conventionally stored in an n-by-n array. In practice, a band linear equation solver would be organized around a data structure that takes advantage of the many zeroes in A. Recall from §1.2.6 that if A has lower bandwidth p and upper bandwidth q it can be represented in a $(p+q+1)$-by-n array *A.band* where band entry a_{ij} is stored in $A.band(i-j+q+1, j)$. In this arrangement, the nonzero portion of A's jth column is housed in the jth column of *A.band*. Another possible band matrix data structure that we discussed in §1.2.8

involves storing A by diagonal in a 1-dimensional array $A.diag$. Regardless of the data structure adopted, the design of a matrix computation with a band storage arrangement requires care in order to minimize subscripting overheads.

Problems

P4.3.1 Derive a banded LDM^T procedure similar to Algorithm 4.3.1.

P4.3.2 Show how the output of Algorithm 4.3.4 can be used to solve the upper Hessenberg system $Hx = b$.

P4.3.3 Give an algorithm for solving an unsymmetric tridiagonal system $Ax = b$ that uses Gaussian elimination with partial pivoting. It should require only four n-vectors of floating point storage for the factorization.

P4.3.4 For $C \in \mathbb{R}^{n \times n}$ define the *profile indices* $m(C,i) = \min\{j : c_{ij} \neq 0\}$, where $i = 1{:}n$. Show that if $A = GG^T$ is the Cholesky factorization of A, then $m(A,i) = m(G,i)$ for $i = 1{:}n$. (We say that G has the same *profile* as A.)

P4.3.5 Suppose $A \in \mathbb{R}^{n \times n}$ is symmetric positive definite with profile indices $m_i = m(A,i)$ where $i = 1{:}n$. Assume that A is stored in a one-dimensional array v as follows: $v = (a_{11}, a_{2,m_2}, \ldots, a_{22}, a_{3,m_3}, \ldots, a_{33}, \ldots, a_{n,m_n}, \ldots, a_{nn})$. Write an algorithm that overwrites v with the corresponding entries of the Cholesky factor G and then uses this factorization to solve $Ax = b$. How many flops are required?

P4.3.6 For $C \in \mathbb{R}^{n \times n}$ define $p(C,i) = \max\{j : c_{ij} \neq 0\}$. Suppose that $A \in \mathbb{R}^{n \times n}$ has an LU factorization $A = LU$ and that:

$$\begin{array}{ccccccc} m(A,1) & \leq & m(A,2) & \leq & \cdots & \leq & m(A,n) \\ p(A,1) & \leq & p(A,2) & \leq & \cdots & \leq & p(A,n) \end{array}$$

Show that $m(A,i) = m(L,i)$ and $p(A,i) = p(U,i)$ for $i = 1{:}n$. Recall the definition of $m(A,i)$ from P4.3.4.

P4.3.7 Develop a gaxpy version of Algorithm 4.3.1.

P4.3.8 Develop a unit stride, vectorizable algorithm for solving the symmetric positive definite tridiagonal systems $A^{(k)}x^{(k)} = b^{(k)}$. Assume that the diagonals, superdiagonals, and right hand sides are stored by row in arrays D, E, and B and that $b^{(k)}$ is overwritten with $x^{(k)}$.

P4.3.9 Develop a version of Algorithm 4.3.1 in which A is stored by diagonal.

P4.3.10 Give an example of a 3-by-3 symmetric positive definite matrix whose tridiagonal part is not positive definite.

P4.3.11 Consider the $Ax = b$ problem where

$$A = \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 & -1 \\ -1 & 2 & -1 & \ddots & \vdots & 0 \\ 0 & -1 & 2 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & \ddots & 2 & -1 \\ -1 & 0 & \cdots & 0 & -1 & 2 \end{bmatrix}$$

This kind of matrix arises in boundary value problems with periodic boundary conditions. (a) Show A is singular. (b) Give conditions that b must satisfy for there to exist a solution and specify an algorithm for solving it. (c). Assume that n is even and consider the permutation

$$P = [\, e_1 \; e_n \; e_2 \; e_{n-1} \; e_3 \; \cdots \,]$$

where e_k is the kth column of I_n. Describe the transformed system $P^TAP(P^Tx) = P^Tb$ and show how to solve it. Assume that there is a solution and ignore pivoting.

Notes and References for Sec. 4.3

The literature concerned with banded systems is immense. Some representative papers include

R.S. Martin and J.H. Wilkinson (1965). "Symmetric Decomposition of Positive Definite Band Matrices," *Numer. Math.* 7, 355–61.

R. S. Martin and J.H. Wilkinson (1967). "Solution of Symmetric and Unsymmetric Band Equations and the Calculation of Eigenvalues of Band Matrices," *Numer. Math.* 9, 279–301.

E.L. Allgower (1973). "Exact Inverses of Certain Band Matrices," *Numer. Math.* 21, 279–84.

Z. Bohte (1975). "Bounds for Rounding Errors in the Gaussian Elimination for Band Systems," *J. Inst. Math. Applic.* 16, 133–42.

I.S. Duff (1977). "A Survey of Sparse Matrix Research," *Proc. IEEE* 65, 500–535.

N.J. Higham (1990). "Bounding the Error in Gaussian Elimination for Tridiagonal Systems," *SIAM J. Matrix Anal. Appl.* 11, 521–530.

A topic of considerable interest in the area of banded matrices deals with methods for reducing the width of the band. See

E. Cuthill (1972). "Several Strategies for Reducing the Bandwidth of Matrices," in *Sparse Matrices and Their Applications*, ed. D.J. Rose and R.A. Willoughby, Plenum Press, New York.

N.E. Gibbs, W.G. Poole, Jr., and P.K. Stockmeyer (1976). "An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix," *SIAM J. Num. Anal.* 13, 236–50.

N.E. Gibbs, W.G. Poole, Jr., and P.K. Stockmeyer (1976). "A Comparison of Several Bandwidth and Profile Reduction Algorithms," *ACM Trans. Math. Soft.* 2, 322–30.

As we mentioned, tridiagonal systems arise with particular frequency. Thus, it is not surprising that a great deal of attention has been focused on special methods for this class of banded problems.

C. Fischer and R.A. Usmani (1969). "Properties of Some Tridiagonal Matrices and Their Application to Boundary Value Problems," *SIAM J. Num. Anal.* 6, 127–42.

D.J. Rose (1969). "An Algorithm for Solving a Special Class of Tridiagonal Systems of Linear Equations," *Comm. ACM* 12, 234–36.

H.S. Stone (1973). "An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations," *J. ACM* 20, 27–38.

M.A. Malcolm and J. Palmer (1974). "A Fast Method for Solving a Class of Tridiagonal Systems of Linear Equations," *Comm. ACM* 17, 14–17.

J. Lambiotte and R.G. Voigt (1975). "The Solution of Tridiagonal Linear Systems of the CDC-STAR 100 Computer," *ACM Trans. Math. Soft.* 1, 308–29.

H.S. Stone (1975). "Parallel Tridiagonal Equation Solvers," *ACM Trans. Math. Soft.* 1, 289–307.

D. Kershaw(1982). "Solution of Single Tridiagonal Linear Systems and Vectorization of the ICCG Algorithm on the Cray-1," in G. Roderigue (ed), *Parallel Computation*, Academic Press, NY, 1982.

N.J. Higham (1986). "Efficient Algorithms for computing the condition number of a tridiagonal matrix," *SIAM J. Sci. and Stat. Comp.* 7, 150–165.

Chapter 4 of George and Liu (1981) contains a nice survey of band methods for positive definite systems.

4.4 Symmetric Indefinite Systems

A symmetric matrix whose quadratic form x^TAx takes on both positive and negative values is called *indefinite*. Although an indefinite A may have an LDL^T factorization, the entries in the factors can have arbitrary magnitude:

$$\begin{bmatrix} \epsilon & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1/\epsilon & 1 \end{bmatrix} \begin{bmatrix} \epsilon & 0 \\ 0 & -1/\epsilon \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1/\epsilon & 1 \end{bmatrix}^T .$$

Of course, any of the pivot strategies in §3.4 could be invoked. However, they destroy symmetry and with it, the chance for a "Cholesky speed" indefinite system solver. Symmetric pivoting, i.e., data reshufflings of the form $A \leftarrow PAP^T$, must be used as we discussed in §4.2.9. Unfortunately, symmetric pivoting does not always stabilize the LDL^T computation. If ϵ_1 *and* ϵ_2 are small then regardless of P, the matrix

$$\tilde{A} = P \begin{bmatrix} \epsilon_1 & 1 \\ 1 & \epsilon_2 \end{bmatrix} P^T$$

has small diagonal entries and large numbers surface in the factorization. With symmetric pivoting, the pivots are always selected from the diagonal and trouble results if these numbers are small relative to what must be zeroed off the diagonal. Thus, LDL^T with symmetric pivoting cannot be recommended as a reliable approach to symmetric indefinite system solving. It seems that the challenge is to involve the off-diagonal entries in the pivoting process while at the same time maintaining symmetry.

In this section we discuss two ways to do this. The first method is due to Aasen(1971) and it computes the factorization

$$PAP^T = LTL^T \qquad (4.4.1)$$

where $L = (\ell_{ij})$ is unit lower triangular and T is tridiagonal. P is a permutation chosen such that $|\ell_{ij}| \leq 1$. In contrast, the *diagonal pivoting method* due to Bunch and Parlett (1971) computes a permutation P such that

$$PAP^T = LDL^T \qquad (4.4.2)$$

where D is a direct sum of 1-by-1 and 2-by-2 pivot blocks. Again, P is chosen so that the entries in the unit lower triangular L satisfy $|\ell_{ij}| \leq 1$. Both factorizations involve $n^3/3$ flops and once computed, can be used to solve $Ax = b$ with $O(n^2)$ work:

$$PAP^T = LTL^T, Lz = Pb, Tw = z, L^Ty = w, x = Py \quad \Rightarrow \quad Ax = b$$

$$PAP^T = LDL^T, Lz = Pb, Dw = z, L^Ty = w, x = Py \quad \Rightarrow \quad Ax = b$$

The only thing "new" to discuss in these solution procedures are the $Tw = z$ and $Dw = z$ systems.

In Aasen's method, the symmetric indefinite tridiagonal system $Tw = z$ is solved in $O(n)$ time using band Gaussian elimination with pivoting. Note that there is no serious price to pay for the disregard of symmetry at this level since the overall process is $O(n^3)$.

In the diagonal pivoting approach, the $Dw = z$ system amounts to a set of 1-by-1 and 2-by-2 symmetric indefinite systems. The 2-by-2 problems can be handled via Gaussian elimination with pivoting. Again, there is no harm in disregarding symmetry during this $O(n)$ phase of the calculation.

Thus, the central issue in this section is the efficient computation of the factorizations (4.4.1) and (4.4.2).

4.4.1 The Parlett-Reid Algorithm

Parlett and Reid (1970) show how to compute (4.4.1) using Gauss transforms. Their algorithm is sufficiently illustrated by displaying the $k = 2$ step for the case $n = 5$. At the beginning of this step the matrix A has been transformed to

$$A^{(1)} = M_1P_1AP_1^TM_1^T = \begin{bmatrix} \alpha_1 & \beta_1 & 0 & 0 & 0 \\ \beta_1 & \alpha_2 & v_3 & v_4 & v_5 \\ 0 & v_3 & \times & \times & \times \\ 0 & v_4 & \times & \times & \times \\ 0 & v_5 & \times & \times & \times \end{bmatrix}$$

where P_1 is a permutation chosen so that the entries in the Gauss transformation M_1 are bounded by unity in modulus. Scanning the vector $(v_3\ v_4\ v_5)^T$ for its largest entry, we now determine a 3-by-3 permutation $\tilde{P}_2$ such that

$$\tilde{P}_2 \begin{bmatrix} v_3 \\ v_4 \\ v_5 \end{bmatrix} = \begin{bmatrix} \tilde{v}_3 \\ \tilde{v}_4 \\ \tilde{v}_5 \end{bmatrix} \quad \Rightarrow \quad |\tilde{v}_3| = \max\{|\tilde{v}_3|, |\tilde{v}_4|, |\tilde{v}_5|\} .$$

If this maximal element is zero, we set $M_2 = P_2 = I$ and proceed to the next step. Otherwise, we set $P_2 = \text{diag}(I_2, \tilde{P}_2)$ and $M_2 = I - \alpha^{(2)}e_3^T$ with

$$\alpha^{(2)} = \begin{pmatrix} 0 & 0 & 0 & \tilde{v}_4/\tilde{v}_3 & \tilde{v}_5/\tilde{v}_3 \end{pmatrix}^T$$

and observe that

$$A^{(2)} = M_2P_2A^{(1)}P_2^TM_2^T = \begin{bmatrix} \alpha_1 & \beta_1 & 0 & 0 & 0 \\ \beta_1 & \alpha_2 & \tilde{v}_3 & 0 & 0 \\ 0 & \tilde{v}_3 & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & \times & \times & \times \end{bmatrix} .$$

In general, the process continues for $n-2$ steps leaving us with a tridiagonal matrix

$$T = A^{(n-2)} = (M_{n-2}P_{n-2}\cdots M_1P_1)A(M_{n-2}P_{n-2}\cdots M_1P_1)^T .$$

It can be shown that (4.4.1) holds with $P = P_{n-2} \cdots P_1$ and

$$L = (M_{n-2}P_{n-2} \cdots M_1 P_1 P^T)^{-1}.$$

Analysis of L reveals that its first column is e_1 and that its subdiagonal entries in column k with $k > 1$ are "made up" of the multipliers in M_{k-1}.

The efficient implementation of the Parlett-Reid method requires care when computing the update

$$A^{(k)} = M_k(P_k A^{(k-1)} P_k^T)M_k^T. \tag{4.4.3}$$

To see what is involved with a minimum of notation, suppose $B = B^T$ has order $n-k$ and that we wish to form: $B_+ = (I - we_1^T)B(I - we_1^T)^T$ where $w \in \mathbb{R}^{n-k}$ and e_1 is the first column of I_{n-k}. Such a calculation is at the heart of (4.4.3). If we set

$$u = Be_1 - \frac{b_{11}}{2}w,$$

then the lower half of the symmetric matrix $B_+ = B - wu^T - uw^T$ can be formed in $2(n-k)^2$ flops. Summing this quantity as k ranges from 1 to $n-2$ indicates that the Parlett-Reid procedure requires $2n^3/3$ flops—twice what we would like.

Example 4.4.1 If the Parlett-Reid algorithm is applied to

$$A = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 2 & 2 \\ 2 & 2 & 3 & 3 \\ 3 & 2 & 3 & 4 \end{bmatrix}$$

then

$$\begin{aligned} P_1 &= [\, e_1\ e_4\ e_3\ e_2 \,] \\ M_1 &= I_4 - (0,\ 0,\ 2/3,\ 1/3,\ )^T e_2^T \\ P_2 &= [\, e_1\ e_2\ e_4\ e_3 \,] \\ M_2 &= I_4 - (0,\ 0,\ 0,\ 1/2)^T e_3^T \end{aligned}$$

and $PAP^T = LTL^T$, where $P = [e_1, e_3, e_4, e_2]$,

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1/3 & 1 & 0 \\ 0 & 2/3 & 1/2 & 1 \end{bmatrix} \quad \text{and} \quad T = \begin{bmatrix} 0 & 3 & 0 & 0 \\ 3 & 4 & 2/3 & 0 \\ 0 & 2/3 & 10/9 & 0 \\ 0 & 0 & 0 & 1/2 \end{bmatrix}.$$

4.4.2 The Method of Aasen

An $n^3/3$ approach to computing (4.4.1) due to Aasen (1971) can be derived by reconsidering some of the computations in the Parlett-Reid approach.

We need a notation for the tridiagonal T:

$$T = \begin{bmatrix} \alpha_1 & \beta_1 & & \cdots & 0 \\ \beta_1 & \alpha_2 & \ddots & & \vdots \\ & \ddots & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & \beta_{n-1} \\ 0 & \cdots & & \beta_{n-1} & \alpha_n \end{bmatrix}.$$

For clarity, we temporarily ignore pivoting and assume that the factorization $A = LTL^T$ exists where L is unit lower triangular with $L(:,1) = e_1$. Aasen's method is organized as follows:

```
for j = 1:n
    Compute h(1:j) where h = TL^T e_j = He_j.
    Compute α(j).
    if j ≤ n − 1
        Compute β(j)                                  (4.4.4)
    end
    if j ≤ n − 2
        Compute L(j + 2:n, j + 1).
    end
end
```

Thus, the mission of the jth Aasen step is to compute the jth column of T and the $(j+1)$-st column of L. The algorithm exploits the fact that the matrix $H = TL^T$ is upper Hessenberg. As can be deduced from (4.4.4), the computation of $\alpha(j)$, $\beta(j)$, and $L(j+2{:}n, j+1)$ hinges upon the vector $h(1{:}j) = H(1{:}j, j)$. Let us see why.

Consider the jth column of the equation $A = LH$:

$$A(:,j) = L(:,1{:}j+1)h(1{:}j+1). \tag{4.4.5}$$

This says that $A(:,j)$ is a linear combination of the first $j+1$ columns of L. In particular,

$$A(j+1{:}n, j) = L(j+1{:}n, 1{:}j)h(1{:}j) + L(j+1{:}n, j+1)h(j+1).$$

It follows that if we compute

$$v(j+1{:}n) = A(j+1{:}n, j) - L(j+1{:}n, 1{:}j)h(1{:}j),$$

then

$$L(j+1{:}n, j+1)h(j+1) = v(j+1{:}n). \tag{4.4.6}$$

Thus, $L(j+2{:}n, j+1)$ is a scaling of $v(j+2{:}n)$. Since L is unit lower triangular we have from (4.4.6) that

$$v(j+1) = h(j+1)$$

and so from that same equation we obtain the following recipe for the $(j+1)$-st column of L:

$$L(j+2{:}n, j+1) = v(j+2{:}n)/v(j+1)\,.$$

Note that $L(j+2{:}n, j+1)$ is a scaled gaxpy.

We next develop formulae for $\alpha(j)$ and $\beta(j)$. Compare the (j,j) and $(j+1,j)$ entries in the equation $H = TL^T$. With the convention $\beta(0) = 0$ we find that $h(j) = \beta(j-1)L(j,j-1) + \alpha(j)$ and $h(j+1) = v(j+1)$ and so

$$\begin{aligned} \alpha(j) &= h(j) - \beta(j-1)L(j,j-1) \\ \beta(j) &= v(j+1)\,. \end{aligned}$$

With these recipes we can completely describe the Aasen procedure:

```
for j = 1:n
    Compute h(1:j) where h = TL^T e_j.
    if j = 1 ∨ j = 2
        α(j) = h(j)
    else
        α(j) = h(j) − β(j − 1)L(j, j − 1)
    end
    if j ≤ n − 1                                          (4.4.7)
        v(j + 1:n) = A(j + 1:n, j) − L(j + 1:n, 1:j)h(1:j)
        β(j) = v(j + 1)
    end
    if j ≤ n − 2
        L(j + 2:n, j + 1) = v(j + 2:n)/v(j + 1)
    end
end
```

To complete the description we must detail the computation of $h(1{:}j)$. From (4.4.5) it follows that

$$A(1{:}j, j) = L(1{:}j, 1{:}j)h(1{:}j)\,. \tag{4.4.8}$$

This lower triangular system can be solved for $h(1{:}j)$ since we know the first j columns of L. However, a much more efficient way to compute $H(1{:}j, j)$ is obtained by exploiting the jth column of the equation $H = TL^T$. In particular, with the convention that $\beta(0)L(j,0) = 0$ we have

$$h(k) = \beta(k-1)L(j,k-1) + \alpha(k)L(j,k) + \beta(k)L(j,k+1)\,.$$

for $k = 1{:}j$. These are working formulae except in the case $k = j$ because we have not yet computed $\alpha(j)$ and $\beta(j)$. However, once $h(1{:}j-1)$ is known we can obtain $h(j)$ from the last row of the triangular system (4.4.8), i.e.,

$$h(j) \;=\; A(j,j) \;-\; \sum_{k=1}^{j-1} L(j,k)h(k)\,.$$

Collecting results and using a work array $\ell(1{:}n)$ for $L(j, 1{:}j)$ we see that the computation of $h(1{:}j)$ in (4.4.7) can be organized as follows:

```
if j = 1
    h(1) = A(1,1)
elseif j = 2
    h(1) = β(1); h(2) = A(2,2)                              (4.4.9)
else
    ℓ(0) = 0; ℓ(1) = 0; ℓ(2:j − 1) = L(j, 2:j − 1); ℓ(j) = 1
    h(j) = A(j, j)
    for k = 1:j − 1
        h(k) = β(k − 1)ℓ(k − 1) + α(k)ℓ(k) + β(k)ℓ(k + 1)
        h(j) = h(j) − ℓ(k)h(k)
    end
end
```

Note that with this $O(j)$ method for computing $h(1{:}j)$, the gaxpy calculation of $v(j+1{:}n)$ is the dominant operation in (4.4.7). During the jth step this gaxpy involves about $2j(n-j)$ flops. Summing this for $j = 1{:}n$ shows that Aasen's method requires $n^3/3$ flops. Thus, the Aasen and Cholesky algorithms entail the same amount of arithmetic.

4.4.3 Pivoting in Aasen's Method

As it now stands, the columns of L are scalings of the v-vectors in (4.4.7). If any of these scalings are large, i.e., if any of the $v(j+1)$'s are small, then we are in trouble. To circumvent this problem we need only permute the largest component of $v(j+1{:}n)$ to the top position. Of course, this permutation must be suitably applied to the unreduced portion of A and the previously computed portion of L.

Algorithm 4.4.1 (Aasen's Method) If $A \in \mathbb{R}^{n\times n}$ is symmetric then the following algorithm computes a permutation P, a unit lower triangular

L, and a tridiagonal T such that $PAP^T = LTL^T$ with $|L(i,j)| \le 1$. The permutation P is encoded in an integer vector piv. In particular, $P = P_1 \cdots P_{n-2}$ where P_j is the identity with rows $piv(j)$ and $j+1$ interchanged. The diagonal and subdiagonal of T are stored in $\alpha(1{:}n)$ and $\beta(1{:}n-1)$, respectively. Only the subdiagonal portion of $L(2{:}n, 2{:}n)$ is computed.

```
for j = 1:n
    Compute h(1:j) via (4.4.9).
    if j = 1 ∨ j = 2
        α(j) = h(j)
    else
        α(j) = h(j) − β(j − 1)L(j, j − 1)
    end
    if j ≤ n − 1
        v(j + 1:n) = A(j + 1:n, j) − L(j + 1:n, 1:j)h(1:j)
        Find q so |v(q)| = ‖ v(j + 1:n) ‖∞ with j + 1 ≤ q ≤ n.
        piv(j) = q; v(j + 1) ↔ v(q); L(j + 1, 2:j) ↔ L(q, 2:j)
        A(j + 1, j + 1:n) ↔ A(q, j + 1:n)
        A(j + 1:n, j + 1) ↔ A(j + 1:n, q)
        β(j) = v(j + 1)
    end
    if j ≤ n − 2
        L(j + 2:n, j + 1) = v(j + 2:n)
        if v(j + 1) ≠ 0
            L(j + 2:n, j + 1) = L(j + 2:n, j + 1)/v(j + 1)
        end
    end
end
```

Aasen's method is stable in the same sense that Gaussian elimination with partial pivoting is stable. That is, the exact factorization of a matrix near A is obtained provided $\| \hat{T} \|_2 / \| A \|_2 \approx 1$, where $\hat{T}$ is the computed version of the tridiagonal matrix T. In general, this is almost always the case.

In a practical implementation of the Aasen algorithm, the lower triangular portion of A would be overwritten with L and T. Here is $n = 5$ case:

$$A \leftarrow \begin{bmatrix} \alpha_1 & & & & \\ \beta_1 & \alpha_2 & & & \\ \ell_{32} & \beta_2 & \alpha_3 & & \\ \ell_{42} & \ell_{43} & \beta_3 & \alpha_4 & \\ \ell_{52} & \ell_{53} & \ell_{54} & \beta_4 & \alpha_5 \end{bmatrix}$$

Notice that the columns of L are shifted left in this arrangement.

4.4.4 Diagonal Pivoting Methods

We next describe the computation of the block LDL^T factorization (4.4.2). We follow the discussion in Bunch and Parlett (1971). Suppose

$$P_1AP_1^T = \begin{bmatrix} E & C^T \\ C & B \end{bmatrix} \begin{matrix} s \\ n-s \end{matrix} \qquad \begin{matrix} s & n-s \end{matrix}$$

where P_1 is a permutation matrix and $s = 1$ or 2. If A is nonzero, then it is always possible to choose these quantities so that E is nonsingular thereby enabling us to write

$$P_1AP_1^T = \begin{bmatrix} I_s & 0 \\ CE^{-1} & I_{n-s} \end{bmatrix} \begin{bmatrix} E & 0 \\ 0 & B - CE^{-1}C^T \end{bmatrix} \begin{bmatrix} I_s & E^{-1}C^T \\ 0 & I_{n-s} \end{bmatrix}$$

For the sake of stability, the s-by-s "pivot" E should be chosen so that the entries in

$$\tilde{A} = (\tilde{a}_{ij}) \equiv B - CE^{-1}C^T \qquad (4.4.10)$$

are suitably bounded. To this end, let $\alpha \in (0,1)$ be given and define the size measures

$$\mu_0 = \max_{i,j} |a_{ij}|$$

$$\mu_1 = \max_{i} |a_{ii}| .$$

The Bunch-Parlett pivot strategy is as follows:

```
if μ1 ≥ αμ0
    s = 1
    Choose P1 so |e11| = μ1.
else
    s = 2
    Choose P1 so |e21| = μ0.
end
```

It is easy to verify from (4.4.10) that if $s = 1$ then

$$|\tilde{a}_{ij}| \le (1 + \alpha^{-1})\mu_0 \qquad (4.4.11)$$

while $s = 2$ implies

$$|\tilde{a}_{ij}| \le \frac{3-\alpha}{1-\alpha}\mu_0 . \qquad (4.4.12)$$

By equating $(1+\alpha^{-1})^2$, the growth factor associated with two $s = 1$ steps, and $(3-\alpha)/(1-\alpha)$, the corresponding $s = 2$ factor, Bunch and Parlett conclude that $\alpha = (1+\sqrt{17})/8$ is optimum from the standpoint of minimizing the bound on element growth.

The reductions outlined above are then repeated on the $n - s$ order symmetric matrix $\tilde{A}$. A simple induction argument establishes that the factorization (4.4.2) exists and that $n^3/3$ flops are required if the work associated with pivot determination is ignored.

4.4.5 Stability and Efficiency

Diagonal pivoting with the above strategy is shown by Bunch (1971) to be as stable as Gaussian elimination with complete pivoting. Unfortunately, the overall process requires between $n^3/12$ and $n^3/6$ comparisons, since μ_0 involves a two-dimensional search at each stage of the reduction. The actual number of comparisons depends on the total number of 2-by-2 pivots but in general the Bunch-Parlett method for computing (4.4.2) is considerably slower than the technique of Aasen. See Barwell and George(1976).

This is not the case with the diagonal pivoting method of Bunch and Kaufman (1977). In their scheme, it is only necessary to scan two columns at each stage of the reduction. The strategy is fully illustrated by considering the very first step in the reduction:

```
α = (1 + √17)/8;  λ = |a_r1| = max{|a_21|, ..., |a_n1|}
if λ > 0
    if |a_11| ≥ αλ
        s = 1; P_1 = I
    else
        σ = |a_pr| = max{|a_1r, ..., |a_{r-1,r}|, |a_{r+1,r}|, ..., |a_nr|}
        if σ|a_11| ≥ αλ^2
            s = 1, P_1 = I
        elseif |a_rr| ≥ ασ
            s = 1 and choose P_1 so (P_1^T A P_1)_11 = a_rr.
        else
            s = 2 and choose P_1 so (P_1^T A P_1)_21 = a_rp.
        end
    end
end
```

Overall, the Bunch-Kaufman algorithm requires $n^3/3$ flops, $O(n^2)$ comparisons, and, like all the methods of this section, $n^2/2$ storage.

Example 4.4.2 If the Bunch-Kaufman algorithm is applied to

$$A = \begin{bmatrix} 1 & 10 & 20 \\ 10 & 1 & 30 \\ 20 & 30 & 1 \end{bmatrix}$$

then in the first step $\lambda = 20$, $r = 3$, $\sigma = 30$, and $p = 2$. The permutation $P = [\, e_3 \; e_2 \; e_1 \,]$ is applied giving

$$PAP^T = \begin{bmatrix} 1 & 30 & 20 \\ 30 & 1 & 10 \\ 20 & 10 & 1 \end{bmatrix}.$$

A 2-by-2 pivot is then used to produce the reduction

$$PAP^T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ .3115 & .6563 & 1 \end{bmatrix} \begin{bmatrix} 1 & 30 & 0 \\ 30 & 1 & 0 \\ 0 & 0 & -11.7920 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ .3115 & .6563 & 1 \end{bmatrix}^T$$

4.4.6 A Note on Equilibrium Systems

A very important class of symmetric indefinite matrices have the form

$$A = \begin{bmatrix} C & B \\ B^T & 0 \end{bmatrix} \begin{matrix} n \\ p \end{matrix} \qquad (4.4.13)$$
$$\quad\; n \quad p$$

where C is symmetric positive definite and B has full column rank. These conditions ensure that A is nonsingular.

Of course, the methods of this section apply to A. However, they do not exploit its structure because the pivot strategies "wipe out" the zero (2,2) block. On the other hand, here is a tempting approach that does exploit A's block structure:

(a) Compute the Cholesky factorization of C, $C = GG^T$.

(b) Solve $GK = B$ for $K \in \mathbb{R}^{n\times p}$.

(c) Compute the Cholesky factorization of $K^TK = B^TC^{-1}B$, $HH^T = K^TK$.

From this it follows that

$$A = \begin{bmatrix} G & 0 \\ K^T & H \end{bmatrix} \begin{bmatrix} G^T & K \\ 0 & -H^T \end{bmatrix}.$$

In principle, this triangular factorization can be used to solve the *equilibrium system*

$$\begin{bmatrix} C & B \\ B^T & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix}. \qquad (4.4.14)$$

However, it is clear by considering steps (b) and (c) above that the accuracy of the computed solution depends upon $\kappa(C)$ and this quantity may be much greater than $\kappa(A)$. The situation has been carefully analyzed and various structure-exploiting algorithms have been proposed. A brief review of the literature is given at the end of the section.

But before we close it is interesting to consider a special case of (4.4.14) that clarifies what it means for an algorithm to be stable and illustrates how perturbation analysis can structure the search for better methods. In several important applications, $g = 0$, C is diagonal, and the solution subvector y is of primary importance. A manipulation of (4.4.14) shows that this vector is specified by

$$y = (B^TC^{-1}B)^{-1}B^TC^{-1}f. \qquad (4.4.15)$$

Looking at this we are again led to believe that $\kappa(C)$ should have a bearing on the accuracy of the computed y. However, it can be shown that

$$\| (B^TC^{-1}B)^{-1}B^TC^{-1} \| \le \psi_B \qquad (4.4.16)$$

where the upper bound ψ_B is independent of C, a result that (correctly) suggests that y is *not* sensitive to perturbations in C. A stable method for computing this vector should respect this, meaning that the accuracy of the computed y should be independent of C. Vavasis (1994) has developed a method with this property. It involves the careful assembly of a matrix $V \in \mathbb{R}^{n\times(n-p)}$ whose columns are a basis for the nullspace of B^TC^{-1}. The n-by-n linear system

$$[\, B\,,\, V\,] \begin{bmatrix} y \\ q \end{bmatrix} = f$$

is then solved implying $f = By + Vq$. Thus, $B^TC^{-1}f = B^TC^{-1}By$ and (4.4.15) holds.

Problems

P4.4.1 Show that if all the 1-by-1 and 2-by-2 principal submatrices of an n-by-n symmetric matrix A are singular, then A is zero.

P4.4.2 Show that no 2-by-2 pivots can arise in the Bunch-Kaufman algorithm if A is positive definite.

P4.4.3 Arrange Algorithm 4.4.1 so that only the lower triangular portion of A is referenced and so that $\alpha(j)$ overwrites $A(j,j)$ for $j = 1{:}n$, $\beta(j)$ overwrites $A(j+1,j)$ for $j = 1{:}n-1$, and $L(i,j)$ overwrites $A(i,j-1)$ for $j = 2{:}n-1$ and $i = j+1{:}n$.

P4.4.4 Suppose $A \in \mathbb{R}^{n\times n}$ is nonsingular, symmetric, and strictly diagonally dominant. Give an algorithm that computes the factorization

$$\Pi A \Pi^T = \begin{bmatrix} R & 0 \\ S & -M \end{bmatrix} \begin{bmatrix} R^T & S^T \\ 0 & M^T \end{bmatrix}$$

where $R \in \mathbb{R}^{k\times k}$ and $M \in \mathbb{R}^{(n-k)\times(n-k)}$ are lower triangular and nonsingular and Π is a permutation.

P4.4.5 Show that if

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & -A_{22} \end{bmatrix} \begin{matrix} n \\ p \end{matrix}$$
$$\qquad\; n \qquad p$$

is symmetric with A_{11} and A_{22} positive definite, then it has an LDL^T factorization with the property that

$$D = \begin{bmatrix} D_1 & 0 \\ 0 & -D_2 \end{bmatrix}$$

where $D_1 \in \mathbb{R}^{n\times n}$ and $D_2 \in \mathbb{R}^{p\times p}$ have positive diagonal entries.

P4.4.6 Prove (4.4.11) and (4.4.12).

P4.4.7 Show that $-(B^TC^{-1}B)^{-1}$ is the (2,2) block of A^{-1} where A is given by (4.4.13).

P4.4.8 The point of this problem is to consider a special case of (4.4.15). Define the matrix

$$M(\alpha) = (B^TC^{-1}B)^{-1}B^TC^{-1}$$

where

$$C = (I_n + \alpha e_k e_k^T) \qquad \alpha > -1.$$

and $e_k = I_n(:,k)$. (Note that C is just the identity with α added to the (k,k) entry.) Assume that $B \in \mathbb{R}^{n\times p}$ has rank p and show that

$$M(\alpha) = (B^TB)^{-1}B^T\left(I_n - \frac{\alpha}{1+\alpha w^Tw}e_k w^T\right)$$

where $w = (I_n - B(B^TB)^{-1}B^T)e_k$. Show that if $\| w \|_2 = 0$ or $\| w \|_2 = 1$, then $\| M(\alpha) \|_2 = 1/\sigma_{min}(B)$. Show that if $0 < \| w \|_2 < 1$, then

$$\| M(\alpha) \|_2 \le \max\left\{ \frac{1}{1 - \| w \|_2}\,,\, 1 + \frac{1}{\| w \|_2} \right\} \Big/ \sigma_{min}(B).$$

Thus, $\| M(\alpha) \|_2$ has an α-independent upper bound.

Notes and References for Sec. 4.4

The basic references for computing (4.4.1) are

J.O. Aasen (1971). "On the Reduction of a Symmetric Matrix to Tridiagonal Form," *BIT 11*, 233–42.

B.N. Parlett and J.K. Reid (1970). "On the Solution of a System of Linear Equations Whose Matrix is Symmetric but not Definite," *BIT 10*, 386–97.

The diagonal pivoting literature includes

J.R. Bunch and B.N. Parlett (1971). "Direct Methods for Solving Symmetric Indefinite Systems of Linear Equations," *SIAM J. Num. Anal. 8*, 639–55.

J.R. Bunch (1971). "Analysis of the Diagonal Pivoting Method," *SIAM J. Num. Anal. 8*, 656–680.

J.R. Bunch (1974). "Partial Pivoting Strategies for Symmetric Matrices," *SIAM J. Num. Anal. 11*, 521–528.

J.R. Bunch, L. Kaufman, and B.N. Parlett (1976). "Decomposition of a Symmetric Matrix," *Numer. Math. 27*, 95–109.

J.R. Bunch and L. Kaufman (1977). "Some Stable Methods for Calculating Inertia and Solving Symmetric Linear Systems," *Math. Comp. 31*, 162–79.

I.S. Duff, N.I.M. Gould, J.K. Reid, J.A. Scott, and K. Turner (1991). "The Factorization of Sparse Indefinite Matrices," *IMA J. Num. Anal. 11*, 181–204.

M.T. Jones and M.L. Patrick (1993). "Bunch-Kaufman Factorization for Real Symmetric Indefinite Banded Matrices," *SIAM J. Matrix Anal. Appl.* *14*, 553–559.

Because "future" columns must be scanned in the pivoting process, it is awkward (but possible) to obtain a gaxpy-rich diagonal pivoting algorithm. On the other hand, Aasen's method is naturally rich in gaxpy's. Block versions of both procedures are possible. LAPACK uses the diagonal pivoting method. Various performance issues are discussed in

V. Barwell and J.A. George (1976). "A Comparison of Algorithms for Solving Symmetric Indefinite Systems of Linear Equations," *ACM Trans. Math. Soft.* *2*, 242–51.
M.T. Jones and M.L. Patrick (1994). "Factoring Symmetric Indefinite Matrices on High-Performance Architectures," *SIAM J. Matrix Anal. Appl.* *15*, 273–283.

Another idea for a cheap pivoting strategy utilizes error bounds based on more liberal interchange criteria, an idea borrowed from some work done in the area of sparse elimination methods. See

R. Fletcher (1976). "Factorizing Symmetric Indefinite Matrices," *Lin. Alg. and Its Applic.* *14*, 257–72.

Before using any symmetric $Ax = b$ solver, it may be advisable to equilibrate A. An $O(n^2)$ algorithm for accomplishing this task is given in

J.R. Bunch (1971). "Equilibration of Symmetric Matrices in the Max-Norm," *J. ACM* *18*, 566–72.

Analogues of the symmetric indefinite solvers that we have presented exist for skew-symmetric systems. See

J.R. Bunch (1982). "A Note on the Stable Decomposition of Skew Symmetric Matrices," *Math. Comp.* *158*, 475–480.

The equilibrium system literature is scattered among the several application areas where it has an important role to play. Nice overviews with pointers to this literature include

G. Strang (1988). "A Framework for Equilibrium Equations," *SIAM Review* *30*, 283–297.
S.A. Vavasis (1994). "Stable Numerical Algorithms for Equilibrium Systems," *SIAM J. Matrix Anal. Appl.* *15*, 1108–1131.

Other papers include

C.C. Paige (1979). "Fast Numerically Stable Computations for Generalized Linear Least Squares Problems," *SIAM J. Num. Anal.* *16*, 165–71.
Å. Björck and I.S. Duff (1980). "A Direct Method for the Solution of Sparse Linear Least Squares Problems," *Lin. Alg. and Its Applic.* *34*, 43–67.
Å. Björck (1992). "Pivoting and Stability in the Augmented System Method," *Proceedings of the 14th Dundee Conference*, D.F. Griffiths and G.A. Watson (eds), Longman Scientific and Technical, Essex, U.K.
P.D. Hough and S.A. Vavasis (1996). "Complete Orthogonal Decomposition for Weighted Least Squares," *SIAM J. Matrix Anal. Appl.*, to appear.

Some of these papers make use of the QR factorization and other least squares ideas that are discussed in the next chapter and §12.1.

Problems with structure abound in matrix computations and perturbation theory has a key role to play in the search for stable, efficient algorithms. For equilibrium systems, there are several results like (4.4.15) that underpin the most effective algorithms. See

A. Forsgren (1995). "On Linear Least-Squares Problems with Diagonally Dominant Weight Matrices," Technical Report TRITA-MAT-1995-OS2, Department of Mathematics, Royal Institute of Technology, S-100 44, Stockholm, Sweden.

and the included references. A discussion of (4.4.15) may be found in

G.W. Stewart (1989). "On Scaled Projections and Pseudoinverses," *Lin. Alg. and Its Applic.* *112*, 189–193.
D.P. O'Leary (1990). "On Bounds for Scaled Projections and Pseudoinverses," *Lin. Alg. and Its Applic.* *132*, 115–117.
M.J. Todd (1990). "A Dantzig-Wolfe-like Variant of Karmarkar's Interior-Point Linear Programming Algorithm," *Operations Research* *38*, 1006–1018.

4.5 Block Systems

In many application areas the matrices that arise have exploitable block structure. As a case study we have chosen to discuss block tridiagonal systems of the form

$$\begin{bmatrix} D_1 & F_1 & & \cdots & 0 \\ E_1 & D_2 & \ddots & & \vdots \\ & \ddots & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & F_{n-1} \\ 0 & \cdots & & E_{n-1} & D_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_n \end{bmatrix} \qquad (4.5.1)$$

Here we assume that all blocks are q-by-q and that the x_i and b_i are in $\mathbb{R}^q$. In this section we discuss both a block LU approach to this problem as well as a divide and conquer scheme known as *cyclic reduction*. Kronecker product systems are briefly mentioned.

4.5.1 Block Tridiagonal LU Factorization

We begin by considering a block LU factorization for the matrix in (4.5.1). Define the block tridiagonal matrices A_k by

$$A_k = \begin{bmatrix} D_1 & F_1 & & \cdots & 0 \\ E_1 & D_2 & \ddots & & \vdots \\ & \ddots & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & F_{k-1} \\ 0 & \cdots & & E_{k-1} & D_k \end{bmatrix} \qquad k = 1{:}n. \qquad (4.5.2)$$

Comparing blocks in

$$A_n = \begin{bmatrix} I & & \cdots & 0 \\ L_1 & I & & \vdots \\ & \ddots & \ddots & \\ \vdots & & \ddots & \\ 0 & \cdots & L_{n-1} & I \end{bmatrix} \begin{bmatrix} U_1 & F_1 & & \cdots & 0 \\ 0 & U_2 & \ddots & & \vdots \\ & \ddots & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & F_{n-1} \\ 0 & \cdots & & 0 & U_n \end{bmatrix} \quad (4.5.3)$$

we formally obtain the following algorithm for the L_i and U_i:

$U_1 = D_1$
for $i = 2{:}n$
  Solve $L_{i-1}U_{i-1} = E_{i-1}$ for L_{i-1}.   (4.5.4)
  $U_i = D_i - L_{i-1}F_{i-1}$
end

The procedure is defined so long as the U_i are nonsingular. This is assured, for example, if the matrices $A_1, \ldots, A_n$ are nonsingular.

Having computed the factorization (4.5.3), the vector x in (4.5.1) can be obtained via block forward and back substitution:

$y_1 = b_1$
for $i = 2{:}n$
  $y_i = b_i - L_{i-1}y_{i-1}$
end   (4.5.5)
Solve $U_n x_n = y_n$ for x_n.
for $i = n-1{:}-1{:}1$
  Solve $U_i x_i = y_i - F_i x_{i+1}$ for x_i.
end

To carry out both (4.5.4) and (4.5.5), each U_i must be factored since linear systems involving these submatrices are solved. This could be done using Gaussian elimination with pivoting. However, this does not guarantee the stability of the overall process. To see this just consider the case when the block size q is unity.

4.5.2 Block Diagonal Dominance

In order to obtain satisfactory bounds on the L_i and U_i it is necessary to make additional assumptions about the underlying block matrix. For example, if for $i = 1{:}n$ we have the block diagonal dominance relations

$$\| D_i^{-1} \|_1 (\| F_{i-1} \|_1 + \| E_i \|_1) < 1 \qquad E_n \equiv F_0 \equiv 0 \quad (4.5.6)$$

then the factorization (4.5.3) exists and it is possible to show that the L_i and U_i satisfy the inequalities

$$\| L_i \|_1 \leq 1 \quad (4.5.7)$$
$$\| U_i \|_1 \leq \| A_n \|_1 \quad (4.5.8)$$

4.5.3 Block Versus Band Solving

At this point it is reasonable to ask why we do not simply regard the matrix A in (4.5.1) as a qn-by-qn matrix having scalar entries and bandwidth $2q-1$. Band Gaussian elimination as described in §4.3 could be applied. The effectiveness of this course of action depends on such things as the dimensions of the blocks and the sparsity patterns within each block.

To illustrate this in a very simple setting, suppose that we wish to solve

$$\begin{bmatrix} D_1 & F_1 \\ E_1 & D_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \quad (4.5.9)$$

where D_1 and D_2 are diagonal and F_1 and E_1 are tridiagonal. Assume that each of these blocks is n-by-n and that it is "safe" to solve (4.5.9) via (4.5.3) and (4.5.5). Note that

$$\begin{aligned} U_1 &= D_1 && \text{(diagonal)} \\ L_1 &= E_1 U_1^{-1} && \text{(tridiagonal)} \\ U_2 &= D_2 - L_1 F_1 && \text{(pentadiagonal)} \\ y_1 &= b_1 \\ y_2 &= b_2 - E_1(D_1^{-1} y_1) \\ U_2 x_2 &= y_2 \\ D_1 x_1 &= y_1 - F_1 x_2. \end{aligned}$$

Consequently, some very simple n-by-n calculations with the original banded blocks renders the solution.

On the other hand, the naive application of band Gaussian elimination to the system (4.5.9) would entail a great deal of unnecessary work and storage as the system has bandwidth $n+1$. However, we mention that by permuting the rows and columns of the system via the permutation

$$P = [e_1, e_{n+1}, e_2, \ldots, e_n, e_{2n}] \quad (4.5.10)$$

we find (in the $n = 5$ case) that

$$PAP^T = \begin{bmatrix} \times & \times & 0 & \times & 0 & 0 & 0 & 0 & 0 & 0 \\ \times & \times & \times & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \times & \times & \times & 0 & \times & 0 & 0 & 0 & 0 \\ \times & 0 & \times & \times & \times & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \times & \times & \times & 0 & \times & 0 & 0 \\ 0 & 0 & \times & 0 & \times & \times & \times & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \times & \times & \times & 0 & \times \\ 0 & 0 & 0 & 0 & \times & 0 & \times & \times & \times & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & 0 & 0 & 0 & \times & 0 & \times & \times \end{bmatrix}$$

This matrix has upper and lower bandwidth equal to three and so a very reasonable solution procedure results by applying band Gaussian elimination to this permuted version of A.

The subject of bandwidth-reducing permutations is important. See George and Liu (1981, Chapter 4). We also refer to the reader to Varah (1972) and George (1974) for further details concerning the solution of block tridiagonal systems.

4.5.4 Block Cyclic Reduction

We next describe the method of *block cyclic reduction* that can be used to solve some important special instances of the block tridiagonal system (4.5.1). For simplicity, we assume that A has the form

$$A = \begin{bmatrix} D & F & & \cdots & 0 \\ F & D & \ddots & & \vdots \\ \cdot & \ddots & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & F \\ 0 & \cdots & & F & D \end{bmatrix} \in \mathbb{R}^{nq \times nq} \qquad (4.5.11)$$

where F and D are q-by-q matrices that satisfy $DF = FD$. We also assume that $n = 2^k - 1$. These conditions hold in certain important applications such as the discretization of Poisson's equation on a rectangle. In that situation,

$$D = \begin{bmatrix} 4 & -1 & & \cdots & 0 \\ -1 & 4 & \ddots & & \vdots \\ & \ddots & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & -1 \\ 0 & \cdots & & -1 & 4 \end{bmatrix} \qquad (4.5.12)$$

and $F = -I_q$. The integer n is determined by the size of the mesh and can often be chosen to be of the form $n = 2^k - 1$. (Sweet (1977) shows how to proceed when the dimension is not of this form.)

The basic idea behind cyclic reduction is to halve the dimension of the problem on hand repeatedly until we are left with a single q-by-q system for the unknown subvector $x_{2^{k-1}}$. This system is then solved by standard means. The previously eliminated x_i are found by a back-substitution process.

The general procedure is adequately motivated by considering the case $n = 7$:

$$\begin{array}{lcl} b_1 & = & Dx_1 + Fx_2 \\ b_2 & = & Fx_1 + Dx_2 + Fx_3 \\ b_3 & = & \qquad Fx_2 + Dx_3 + Fx_4 \\ b_4 & = & \qquad\qquad Fx_3 + Dx_4 + Fx_5 \\ b_5 & = & \qquad\qquad\qquad Fx_4 + Dx_5 + Fx_6 \\ b_6 & = & \qquad\qquad\qquad\qquad Fx_5 + Dx_6 + Fx_7 \\ b_7 & = & \qquad\qquad\qquad\qquad\qquad Fx_6 + Dx_7 \end{array} \qquad (4.5.13)$$

For $i = 2$, 4, and 6 we multiply equations $i-1$, i, and $i+1$ by F, $-D$, and F, respectively, and add the resulting equations to obtain

$$\begin{array}{rcl} (2F^2 - D^2)x_2 + F^2x_4 & = & F(b_1 + b_3) - Db_2 \\ F^2x_2 + (2F^2 - D^2)x_4 + F^2x_6 & = & F(b_3 + b_5) - Db_4 \\ F^2x_4 + (2F^2 - D^2)x_6 & = & F(b_5 + b_7) - Db_6 \end{array}$$

Thus, with this tactic we have removed the odd-indexed x_i and are left with a reduced block tridiagonal system of the form

$$\begin{array}{rcl} D^{(1)}x_2 + F^{(1)}x_4 & = & b_2^{(1)} \\ F^{(1)}x_2 + D^{(1)}x_4 + F^{(1)}x_6 & = & b_4^{(1)} \\ F^{(1)}x_4 + D^{(1)}x_6 & = & b_6^{(1)} \end{array}$$

where $D^{(1)} = 2F^2 - D^2$ and $F^{(1)} = F^2$ commute. Applying the same elimination strategy as above, we multiply these three equations respectively by $F^{(1)}$, $-D^{(1)}$, and $F^{(1)}$. When these transformed equations are added together, we obtain the single equation

$$\left(2[F^{(1)}]^2 - D^{(1)^2}\right) x_4 = F^{(1)}\left(b_2^{(1)} + b_6^{(1)}\right) - D^{(1)}b_4^{(1)}$$

which we write as

$$D^{(2)}x_4 = b^{(2)}.$$

This completes the cyclic reduction. We now solve this (small) q-by-q system for x_4. The vectors x_2 and x_6 are then found by solving the systems

$$\begin{array}{rcl} D^{(1)}x_2 & = & b_2^{(1)} - F^{(1)}x_4 \\ D^{(1)}x_6 & = & b_6^{(1)} - F^{(1)}x_4 \end{array}$$

Finally, we use the first, third, fifth, and seventh equations in (4.5.13) to compute x_1, x_3, x_5, and x_7, respectively.

For general n of the form $n = 2^k - 1$ we set $D^{(0)} = D$, $F^{(0)} = F$, $b^{(0)} = b$ and compute:

$$
\begin{array}{l}
\textbf{for } p = 1{:}k-1 \\
\quad F^{(p)} = [F^{(p-1)}]^2 \\
\quad D^{(p)} = 2F^{(p)} - [D^{(p-1)}]^2 \\
\quad r = 2^p \\
\quad \textbf{for } j = 1{:}2^{k-p} - 1 \\
\quad\quad b^{(p)}_{jr} = F^{(p-1)}\left(b^{(p-1)}_{jr-r/2} + b^{(p-1)}_{jr+r/2}\right) - D^{(p-1)}b^{(p-1)}_{jr} \\
\quad \textbf{end} \\
\textbf{end}
\end{array}
\qquad (4.5.14)
$$

The x_i are then computed as follows:

$$
\begin{array}{l}
\text{Solve } D^{(k-1)}x_{2^{k-1}} = b^{(k-1)}_1 \text{ for } x_{2^{k-1}}. \\
\textbf{for } p = k-2{:}-1{:}0 \\
\quad r = 2^p \\
\quad \textbf{for } j = 1{:}2^{k-p-1} \\
\quad\quad \textbf{if } j = 1 \\
\quad\quad\quad c = b^{(p)}_{(2j-1)r} - F^{(p)}x_{2jr} \\
\quad\quad \textbf{elseif } j = 2^{k-p+1} \\
\quad\quad\quad c = b^{(p)}_{(2j-1)r} - F^{(p)}x_{(2j-2)r} \\
\quad\quad \textbf{else} \\
\quad\quad\quad c = b^{(p)}_{(2j-1)r} - F^{(p)}\left(x_{2jr} + x_{(2j-2)r}\right) \\
\quad\quad \textbf{end} \\
\quad\quad \text{Solve } D^{(p)}x_{(2j-1)r} = c \text{ for } x_{(2j-1)r} \\
\quad \textbf{end} \\
\textbf{end}
\end{array}
\qquad (4.5.15)
$$

The amount of work required to perform these recursions depends greatly upon the sparsity of the $D^{(p)}$ and $F^{(p)}$. In the worse case when these matrices are full, the overall flop count has order $\log(n)q^3$. Care must be exercised in order to ensure stability during the reduction. For further details, see Buneman (1969).

Example 4.5.1 Suppose $q = 1$, $D = (4)$, and $F = (-1)$ in (4.5.14) and that we wish to solve:

$$
\begin{bmatrix} 4 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 4 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 4 \end{bmatrix}
\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix}
=
\begin{bmatrix} 2 \\ 4 \\ 6 \\ 8 \\ 10 \\ 12 \\ 22 \end{bmatrix}
$$

By executing (4.5.15) we obtain the reduced systems:

$$
\begin{bmatrix} -14 & 1 & 0 \\ 1 & -14 & 1 \\ 0 & 1 & -14 \end{bmatrix}
\begin{bmatrix} x_2 \\ x_4 \\ x_6 \end{bmatrix}
=
\begin{bmatrix} -24 \\ -48 \\ -80 \end{bmatrix}
\qquad p = 1
$$

and

$$
\begin{bmatrix} -194 \end{bmatrix} = \begin{bmatrix} x_4 \end{bmatrix}\begin{bmatrix} -776 \end{bmatrix} \qquad p = 2
$$

The x_i are then determined via (4.5.16):

$$
\begin{array}{llll}
p = 2{:} & x_4 = 4 & & \\
p = 1{:} & x_2 = 2 & x_6 = 6 & \\
p = 0{:} & x_1 = 1 & x_3 = 3 & x_5 = 5 \quad x_7 = 7
\end{array}
$$

Cyclic reduction is an example of a divide and conquer algorithm. Other divide and conquer procedures are discussed in §1.3.8 and §8.6.

4.5.5 Kronecker Product Systems

If $B \in \mathbb{R}^{m\times n}$ and $C \in \mathbb{R}^{p\times q}$, then their *Kronecker product* is given by

$$
A = B \otimes C = \begin{bmatrix} b_{11}C & b_{12}C & \cdots & b_{1n}C \\ b_{21}C & b_{22}C & \cdots & b_{2n}C \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1}C & b_{m2}C & \cdots & b_{mn}C \end{bmatrix}.
$$

Thus, A is an m-by-n block matrix whose (i,j) block is $b_{ij}C$. Kronecker products arise in conjunction with various mesh discretizations and throughout signal processing. Some of the more important properties that the Kronecker product satisfies include

$$
\begin{aligned}
(A\otimes B)(C\otimes D) &= = AC\otimes BD && (4.5.16)\\
(A\otimes B)^T &= A^T\otimes B^T && (4.5.17)\\
(A\otimes B)^{-1} &= A^{-1}\otimes B^{-1} && (4.5.18)
\end{aligned}
$$

where it is assumed that all the factor operations are defined.

Related to the Kronecker product is the "vec" operation:

$$
X \in \mathbb{R}^{m\times n} \quad \Leftrightarrow \quad \text{vec}(X) = \begin{bmatrix} X(:,1) \\ \vdots \\ X(:,n) \end{bmatrix} \in \mathbb{R}^{mn}.
$$

Thus, the vec of a matrix amounts to a "stacking" of its columns. It can be shown that

$$
Y = CXB^T \quad \Leftrightarrow \quad \text{vec}(Y) = (B\otimes C)\text{vec}(X). \qquad (4.5.19)
$$

It follows that solving a Kronecker product system,

$$(B \otimes C)x = d$$

is equivalent to solving the matrix equation $CXB^T = D$ for X where $x = \text{vec}(X)$ and $d = \text{vec}(D)$. This has efficiency ramifications. To illustrate, suppose $B, C \in \mathbb{R}^{n \times n}$ are symmetric positive definite. If $A = B \otimes C$ is treated as a general matrix and factored in order to solve for x, then $O(n^6)$ flops are required since $B \otimes C \in \mathbb{R}^{n^2 \times n^2}$. On the other hand, the solution approach

1. Compute the Cholesky factorizations $B = GG^T$ and $C = HH^T$.
2. Solve $BZ = D^T$ for Z using G.
3. Solve $CX = Z^T$ for X using H.
4. $x = \text{vec}(X)$.

involves $O(n^3)$ flops. Note that

$$B \otimes C = GG^T \otimes HH^T = (G \otimes H)(G \otimes H)^T$$

is the Cholesky factorization of $B \otimes C$ because the Kronecker product of a pair of lower triangular matrices is lower triangular. Thus, the above four-step solution approach is a structure-exploiting, Cholesky method applied to $B \otimes C$.

We mention that if B is sparse, then $B \otimes C$ has the same sparsity at the block level. For example, if B is tridiagonal, then $B \otimes C$ is block tridiagonal.

Problems

P4.5.1 Show that a block diagonally dominant matrix is nonsingular.

P4.5.2 Verify that (4.5.6) implies (4.5.7) and (4.5.8).

P4.5.3 Suppose block cyclic reduction is applied with D given by (4.5.12) and $F = -I_q$. What can you say about the band structure of the matrices $F^{(p)}$ and $D^{(p)}$ that arise?

P4.5.4 Suppose $A \in \mathbb{R}^{n \times n}$ is nonsingular and that we have solutions to the linear systems $Ax = b$ and $Ay = g$ where $b, g \in \mathbb{R}^n$ are given. Show how to solve the system

$$\begin{bmatrix} A & g \\ h^T & \alpha \end{bmatrix} \begin{bmatrix} x \\ \mu \end{bmatrix} = \begin{bmatrix} b \\ \beta \end{bmatrix}$$

in $O(n)$ flops where $\alpha, \beta \in \mathbb{R}$ and $h \in \mathbb{R}^n$ are given and the matrix of coefficients A_+ is nonsingular. The advisability of going for such a quick solution is a complicated issue that depends upon the condition numbers of A and A_+ and other factors.

P4.5.5 Verify (4.5.16)-(4.5.19).

P4.5.7 Show how to construct the SVD of $B \otimes C$ from the SVDs of B and C.

P4.5.8 If A, B, and C are matrices, then it can be shown that $(A \otimes B) \otimes C = A \otimes (B \otimes C)$

and so we just write $A \otimes B \otimes C$ for this matrix. Show how to solve the linear system $(A \otimes B \otimes C)x = d$ assuming that A, B, and C are symmetric positive definite.

Notes and References for Sec. 4.5

The following papers provide insight into the various nuances of block matrix computations:

J.M. Varah (1972). "On the Solution of Block-Tridiagonal Systems Arising from Certain Finite-Difference Equations," *Math. Comp. 26*, 859–68.
J.A. George (1974). "On Block Elimination for Sparse Linear Systems," *SIAM J. Num. Anal. 11*, 585–603.
R. Fourer (1984). "Staircase Matrices and Systems," *SIAM Review 26*, 1-71.
M.L. Merriam (1985). "On the Factorization of Block Tridiagonals With Storage Constraints," *SIAM J. Sci. and Stat. Comp. 6*, 182-192.

The property of block diagonal dominance and its various implications is the central theme in

D.G. Feingold and R.S. Varga (1962). "Block Diagonally Dominant Matrices and Generalizations of the Gershgorin Circle Theorem," *Pacific J. Math. 12*, 1241–50.

Early methods that involve the idea of cyclic reduction are described in

R.W. Hockney (1965). "A Fast Direct Solution of Poisson's Equation Using Fourier Analysis," *J. ACM 12*, 95-113.
B.L. Buzbee, G.H. Golub, and C.W. Nielson (1970). "On Direct Methods for Solving Poisson's Equations," *SIAM J. Num. Anal. 7*, 627-56.

The accumulation of the right-hand side must be done with great care, for otherwise there would be a significant loss of accuracy. A stable way of doing this is described in

O. Buneman (1969). "A Compact Non-Iterative Poisson Solver," Report 294, Stanford University Institute for Plasma Research, Stanford, California.

Other literature concerned with cyclic reduction includes

F.W. Dorr (1970). "The Direct Solution of the Discrete Poisson Equation on a Rectangle," *SIAM Review 12*, 248–63.
B.L. Buzbee, F.W. Dorr, J.A. George, and G.H. Golub (1971). "The Direct Solution of the Discrete Poisson Equation on Irregular Regions," *SIAM J. Num. Anal. 8*, 722–36.
F.W. Dorr (1973). "The Direct Solution of the Discrete Poisson Equation in $O(n^2)$ Operations," *SIAM Review 15*, 412–415.
P. Concus and G.H. Golub (1973). "Use of Fast Direct Methods for the Efficient Numerical Solution of Nonseparable Elliptic Equations," *SIAM J. Num. Anal. 10*, 1103–20.
B.L. Buzbee and F.W. Dorr (1974). "The Direct Solution of the Biharmonic Equation on Rectangular Regions and the Poisson Equation on Irregular Regions," *SIAM J. Num. Anal. 11*, 753–63.
D. Heller (1976). "Some Aspects of the Cyclic Reduction Algorithm for Block Tridiagonal Linear Systems," *SIAM J. Num. Anal. 13*, 484–96.

Various generalizations and extensions to cyclic reduction have been proposed:

P.N. Swarztrauber and R.A. Sweet (1973). "The Direct Solution of the Discrete Poisson Equation on a Disk," *SIAM J. Num. Anal. 10*, 900–907.

R.A. Sweet (1974). "A Generalized Cyclic Reduction Algorithm," *SIAM J. Num. Anal. 11*, 506–20.
M.A. Diamond and D.L.V. Ferreira (1976). "On a Cyclic Reduction Method for the Solution of Poisson's Equation," *SIAM J. Num. Anal. 13*, 54–70.
R.A. Sweet (1977). "A Cyclic Reduction Algorithm for Solving Block Tridiagonal Systems of Arbitrary Dimension," *SIAM J. Num. Anal. 14*, 706–20.
P.N. Swarztrauber and R. Sweet (1989). "Vector and Parallel Methods for the Direct Solution of Poisson's Equation," *J. Comp. Appl. Math. 27*, 241–263.
S. Bondeli and W. Gander (1994). "Cyclic Reduction for Special Tridiagonal Systems," *SIAM J. Matrix Anal. Appl. 15*, 321–330.

For certain matrices that arise in conjunction with elliptic partial differential equations, block elimination corresponds to rather natural operations on the underlying mesh. A classical example of this is the method of nested dissection described in

A. George (1973). "Nested Dissection of a Regular Finite Element Mesh," *SIAM J. Num. Anal. 10*, 345–63.

We also mention the following general survey:

J.R. Bunch (1976). "Block Methods for Solving Sparse Linear Systems," in *Sparse Matrix Computations*, J.R. Bunch and D.J. Rose (eds), Academic Press, New York.

Bordered linear systems as presented in P4.5.4 are discussed in

W. Govaerts and J.D. Pryce (1990). "Block Elimination with One Iterative Refinement Solves Bordered Linear Systems Accurately," *BIT 30*, 490–507.
W. Govaerts (1991). "Stable Solvers and Block Elimination for Bordered Systems," *SIAM J. Matrix Anal. Appl. 12*, 469–483.
W. Govaerts and J.D. Pryce (1993). "Mixed Block Elimination for Linear Systems with Wider Borders," *IMA J. Num. Anal. 13*, 161–180.

Kronecker product references include

H.C. Andrews and J. Kane (1970). "Kronecker Matrices, Computer Implementation, and Generalized Spectra," *J. Assoc. Comput. Mach. 17*, 260–268.
C. de Boor (1979). "Efficient Computer Manipulation of Tensor Products," *ACM Trans. Math. Soft. 5*, 173–182.
A. Graham (1981). *Kronecker Products and Matrix Calculus with Applications*, Ellis Horwood Ltd., Chichester, England.
H.V. Henderson and S.R. Searle (1981). "The Vec-Permutation Matrix, The Vec Operator, and Kronecker Products: A Review," *Linear and Multilinear Algebra 9*, 271–288.
P.A. Regalia and S. Mitra (1989). "Kronecker Products, Unitary Matrices, and Signal Processing Applications," *SIAM Review 31*, 586–613.

4.6 Vandermonde Systems and the FFT

Suppose $x(0{:}n) \in \mathbb{R}^{n+1}$. A matrix $V \in \mathbb{R}^{(n+1)\times(n+1)}$ of the form

$$V = V(x_0,\ldots,x_n) = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_0 & x_1 & \cdots & x_n \\ \vdots & \vdots & & \vdots \\ x_0^n & x_1^n & \cdots & x_n^n \end{bmatrix}$$

is said to be a *Vandermonde matrix*. In this section, we show how the systems $V^T a = f = f(0{:}n)$ and $Vz = b = b(0{:}n)$ can be solved in $O(n^2)$ flops. The discrete Fourier transform is briefly introduced. This special and extremely important Vandermonde system has a a recursive block structure and can be solved in $O(n \log n)$ flops. *In this section, vectors and matrices are subscripted from 0.*

4.6.1 Polynomial Interpolation: $V^T a = f$

Vandermonde systems arise in many approximation and interpolation problems. Indeed, the key to obtaining a fast Vandermonde solver is to recognize that solving $V^T a = f$ is equivalent to polynomial interpolation. This follows because if $V^T a = f$ and

$$p(x) = \sum_{j=0}^{n} a_j x^j \tag{4.6.1}$$

then $p(x_i) = f_i$ for $i = 0{:}n$.

Recall that if the x_i are distinct then there is a unique polynomial of degree n that interpolates $(x_0, f_0), \ldots, (x_n, f_n)$. Consequently, V is nonsingular as long as the x_i are distinct. We assume this throughout the section.

The first step in computing the a_j of (4.6.1) is to calculate the Newton representation of the interpolating polynomial p:

$$p(x) = \sum_{k=0}^{n} c_k \left(\prod_{i=0}^{k-1} (x - x_i) \right). \tag{4.6.2}$$

The constants c_k are divided differences and may be determined as follows:

$$\begin{array}{l} c(0{:}n) = f(0{:}n) \\ \textbf{for } k = 0{:}n-1 \\ \quad \textbf{for } i = n{:}-1{:}k+1 \\ \qquad c_i = (c_i - c_{i-1})/(x_i - x_{i-k-1}) \\ \quad \textbf{end} \\ \textbf{end} \end{array} \tag{4.6.3}$$

See Conte and de Boor (1980, chapter 2).

The next task is to generate $a(0{:}n)$ from $c(0{:}n)$. Define the polynomials $p_n(x), \ldots, p_0(x)$ by the iteration

$$\begin{array}{l} p_n(x) = c_n \\ \textbf{for } k = n-1{:}-1{:}0 \\ \quad p_k(x) = c_k + (x - x_k)p_{k+1}(x) \\ \textbf{end} \end{array}$$

and observe that $p_0(x) = p(x)$. Writing

$$p_k(x) = a_k^{(k)} + a_{k+1}^{(k)}x + \cdots + a_n^{(k)}x^{n-k}$$

and equating like powers of x in the equation $p_k = c_k + (x - x_k)p_{k+1}$ gives the following recursion for the coefficients $a_i^{(k)}$:

$$\begin{array}{l} a_n^{(n)} = c_n \\ \text{for } k = n-1:-1:0 \\ \quad a_k^{(k)} = c_k - x_k a_{k+1}^{(k+1)} \\ \quad \text{for } i = k+1:n-1 \\ \qquad a_i^{(k)} = a_i^{(k+1)} - x_k a_{i+1}^{(k+1)} \\ \quad \text{end} \\ \quad a_n^{(k)} = a_n^{(k+1)} \\ \text{end} \end{array}$$

Consequently, the coefficients $a_i = a_i^{(0)}$ can be calculated as follows:

$$\begin{array}{l} a(0:n) = c(0:n) \\ \text{for } k = n-1:-1:0 \\ \quad \text{for } i = k:n-1 \\ \qquad a_i = a_i - x_k a_{i+1} \\ \quad \text{end} \\ \text{end} \end{array} \qquad (4.6.4)$$

Combining this iteration with (4.6.3) renders the following algorithm:

Algorithm 4.6.1 Given $x(0:n) \in \mathbb{R}^{n+1}$ with distinct entries and $f = f(0:n) \in \mathbb{R}^{n+1}$, the following algorithm overwrites f with the solution $a = a(0:n)$ to the Vandermonde system $V(x_0, \ldots, x_n)^T a = f$.

$$\begin{array}{l} \text{for } k = 0:n-1 \\ \quad \text{for } i = n:-1:k+1 \\ \qquad f(i) = (f(i) - f(i-1))/(x(i) - x(i-k-1)) \\ \quad \text{end} \\ \text{end} \\ \text{for } k = n-1:-1:0 \\ \quad \text{for } i = k:n-1 \\ \qquad f(i) = f(i) - f(i+1)x(k) \\ \quad \text{end} \\ \text{end} \end{array}$$

This algorithm requires $5n^2/2$ flops.

Example 4.6.1 Suppose Algorithm 4.6.1 is used to solve

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \\ 1 & 3 & 9 & 27 \\ 1 & 4 & 16 & 64 \end{bmatrix}^T \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 10 \\ 26 \\ 58 \\ 112 \end{bmatrix}.$$

The first k-loop computes the Newton representation of $p(x)$:

$$p(x) = 10 + 16(x-1) + 8(x-1)(x-2) + (x-1)(x-2)(x-3).$$

The second k-loop computes $a = [4\ 3\ 2\ 1]^T$ from $[10\ 16\ 8\ 1]^T$.

4.6.2 The System Vz = b

Now consider the system $Vz = b$. To derive an efficient algorithm for this problem, we describe what Algorithm 4.6.1 does in matrix-vector language. Define the lower bidiagonal matrix $L_k(\alpha) \in \mathbb{R}^{(n+1)\times(n+1)}$ by

$$L_k(\alpha) = \left[\begin{array}{c|ccccc} I_k & & & 0 & & \\ \hline & 1 & & \cdots & & 0 \\ & -\alpha & 1 & & & \\ & & \ddots & \ddots & & \\ 0 & \vdots & & \ddots & \ddots & \vdots \\ & & & & \ddots & 1 \\ & 0 & & \cdots & -\alpha & 1 \end{array}\right]$$

and the diagonal matrix D_k by

$$D_k = \text{diag}(\underbrace{1, \ldots, 1}_{k+1}, x_{k+1} - x_0, \ldots, x_n - x_{n-k-1}).$$

With these definitions it is easy to verify from (4.6.3) that if $f = f(0:n)$ and $c = c(0:n)$ is the vector of divided differences then $c = U^T f$ where U is the upper triangular matrix defined by

$$U^T = D_{n-1}^{-1} L_{n-1}(1) \cdots D_0^{-1} L_0(1).$$

Similarly, from (4.6.4) we have

$$a = L^T c,$$

where L is the unit lower triangular matrix defined by:

$$L^T = L_0(x_0)^T \cdots L_{n-1}(x_{n-1})^T.$$

Thus, $a = L^T U^T f$ where $V^{-T} = L^T U^T$. In other words, Algorithm 4.6.1 solves $V^T a = f$ by tacitly computing the "UL" factorization of V^{-1}.

Consequently, the solution to the system $Vz = b$ is given by

$$\begin{aligned} z &= V^{-1}b = U(Lb) \\ &= \left(L_0(1)^T D_0^{-1} \cdots L_{n-1}(1)^T D_{n-1}^{-1}\right)\left(L_{n-1}(x_{n-1}) \cdots L_0(x_0)b\right) \end{aligned}$$

This observation gives rise to the following algorithm:

Algorithm 4.6.2 Given $x(0{:}n) \in \mathbb{R}^{n+1}$ with distinct entries and $b = b(0{:}n) \in \mathbb{R}^{n+1}$, the following algorithm overwrites b with the solution $z = z(0{:}n)$ to the Vandermonde system $V(x_0, \ldots, x_n)z = b$.

```
for k = 0:n - 1
    for i = n: - 1:k + 1
        b(i) = b(i) - x(k)b(i - 1)
    end
end
for k = n - 1: - 1:0
    for i = k + 1:n
        b(i) = b(i)/(x(i) - x(i - k - 1))
    end
    for i = k:n - 1
        b(i) = b(i) - b(i + 1)
    end
end
```

This algorithm requires $5n^2/2$ flops.

Example 4.6.2 Suppose Algorithm 4.6.2 is used to solve

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \\ 1 & 4 & 9 & 16 \\ 1 & 8 & 27 & 64 \end{bmatrix} \begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \\ 3 \\ 35 \end{bmatrix}.$$

The first k-loop computes the vector

$$L_3(3)L_2(2)L_1(1) \begin{bmatrix} 0 \\ -1 \\ 3 \\ 35 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \\ 6 \\ 6 \end{bmatrix}.$$

The second k-loop then calculates

$$L_0(1)^T D_0^{-1} L_1(1)^T D_1^{-1} L_2(1)^T D_2^{-1} \begin{bmatrix} 0 \\ -1 \\ 3 \\ 35 \end{bmatrix} = \begin{bmatrix} 3 \\ -4 \\ 0 \\ 1 \end{bmatrix}.$$

4.6.3 Stability

Algorithms 4.6.1 and 4.6.2 are discussed and analyzed in Björck and Pereyra (1970). Their experience is that these algorithms frequently produce surprisingly accurate solutions, even when V is ill-conditioned. They also show how to update the solution when a new coordinate pair (x_{n+1}, f_{n+1})

is added to the set of points to be interpolated, and how to solve *confluent Vandermonde systems*, i.e., systems involving matrices like

$$V = V(x_0, x_1, x_1, x_3) = \begin{bmatrix} 1 & 1 & 0 & 1 \\ x_0 & x_1 & 1 & x_3 \\ x_0^2 & x_1^2 & 2x_1 & x_3^2 \\ x_0^3 & x_1^3 & 3x_1^2 & x_3^3 \end{bmatrix}.$$

4.6.4 The Fast Fourier Transform

The discrete Fourier transform (DFT) matrix of order n is defined by

$$F_n = (f_{jk}) \qquad f_{jk} = \omega_n^{jk}$$

where

$$\omega_n = \exp(-2\pi i/n) = \cos(2\pi/n) - i \cdot \sin(2\pi/n).$$

The parameter ω_n is an nth root of unity because $\omega_n^n = 1$. In the $n = 4$ case, $\omega_4 = -i$ and

$$F_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_4 & \omega_4^2 & \omega_4^3 \\ 1 & \omega_4^2 & \omega_4^4 & \omega_4^6 \\ 1 & \omega_4^3 & \omega_4^6 & \omega_4^9 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}.$$

If $x \in \mathbb{C}^n$, then its DFT is the vector $F_n x$. The DFT has an extremely important role to play throughout applied mathematics and engineering.

If n is highly composite, then it is possible to carry out the DFT in many fewer than the $O(n^2)$ flops required by conventional matrix-vector multiplication. To illustrate this we set $n = 2^t$ and proceed to develop the *radix-2 fast Fourier transform (FFT)*. The starting point is to look at an even-order DFT matrix when we permute its columns so that the even-indexed columns come first. Consider the case $n = 8$. Noting that $\omega_8^{kj} = \omega_n^{kj \bmod 8}$ we have

$$F_8 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ 1 & \omega^2 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega & \omega^6 & \omega^3 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega \end{bmatrix}, \qquad \omega = \omega_8.$$

If we define the index vector $c = [0\ 2\ 4\ 6\ 1\ 3\ 5\ 7]$, then

$$F_8(:,c) = \left[\begin{array}{cccc|cccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^2 & \omega^4 & \omega^6 & \omega & \omega^3 & \omega^5 & \omega^7 \\ 1 & \omega^4 & 1 & \omega^4 & \omega^2 & \omega^6 & \omega^2 & \omega^6 \\ 1 & \omega^6 & \omega^4 & \omega^2 & \omega^3 & \omega & \omega^7 & \omega^5 \\ \hline 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & \omega^2 & \omega^4 & \omega^6 & -\omega & -\omega^3 & -\omega^5 & -\omega^7 \\ 1 & \omega^4 & 1 & \omega^4 & -\omega^2 & -\omega^6 & -\omega^2 & -\omega^6 \\ 1 & \omega^6 & \omega^4 & \omega^2 & -\omega^3 & -\omega & -\omega^7 & -\omega^5 \end{array}\right].$$

The lines through the matrix are there to help us think of $F_n(:,c)$ as a 2-by-2 matrix with 4-by-4 blocks. Noting that $\omega^2 = \omega_8^2 = \omega_4$ we see that

$$F_8(:,c) = \left[\begin{array}{c|c} F_4 & \Omega_4 F_4 \\ \hline F_4 & -\Omega_4 F_4 \end{array}\right]$$

where

$$\Omega_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \omega_8 & 0 & 0 \\ 0 & 0 & \omega_8^2 & 0 \\ 0 & 0 & 0 & \omega_8^3 \end{bmatrix}.$$

It follows that if x in an 8-vector, then

$$\begin{aligned} F_8 x = F(:,c)x(c) &= \left[\begin{array}{c|c} F_4 & \Omega_4 F_4 \\ \hline F_4 & -\Omega_4 F_4 \end{array}\right] \left[\begin{array}{c} x(0{:}2{:}7) \\ \hline x(1{:}2{:}7) \end{array}\right] \\ &= \left[\begin{array}{c|c} I & \Omega_4 \\ \hline I & -\Omega_4 \end{array}\right] \left[\begin{array}{c} F_4 x(0{:}2{:}7) \\ \hline F_4 x(1{:}2{:}7) \end{array}\right]. \end{aligned}$$

Thus, by simple scalings we can obtain the 8-point DFT $y = F_8 x$ from the 4-point DFTs $y_T = F_4 x(0{:}2{:}7)$ and $y_B = F_4 x(1{:}2{:}7)$:

$$\begin{aligned} y(0{:}4) &= y_T + d.*y_B \\ y(4{:}7) &= y_T - d.*y_B. \end{aligned}$$

Here,

$$d = \begin{bmatrix} 1 \\ \omega \\ \omega^2 \\ \omega^3 \end{bmatrix}$$

and ".*" indicates vector multiplication. In general, if $n = 2m$, then $y = F_n x$ is given by

$$\begin{aligned} y(0{:}m-1) &= y_T + d.*y_B \\ y(m{:}n-1) &= y_B - d.*y_B \end{aligned}$$

where

$$\begin{aligned} d &= \left[\, 1,\ \omega, \cdots,\ \omega^{m-1} \,\right]^T \\ y_T &= F_m x(0{:}2{:}n-1), \\ y_B &= F_m x(1{:}2{:}n-1). \end{aligned}$$

For $n = 2^t$ we can recur on this process until $n = 1$ for which $F_1 x = x$:

function $y = FFT(x, n)$
  if $n = 1$
    $y = x$
  else
    $m = n/2;\ \omega = e^{-2\pi i/n}$
    $y_T = FFT(x(0{:}2{:}n), m);\ y_B = FFT(x(1{:}2{:}n), m)$
    $d = \left[\, 1,\ \omega, \cdots,\ \omega^{m-1} \,\right]^T;\ z = d.*y_B$
    $y = \begin{bmatrix} y_T + z \\ y_T - z \end{bmatrix}$
  end

This is a member of the fast Fourier transform family of algorithms. It has a nonrecursive implementation that is best presented in terms of a factorization of F_n. Indeed, it can be shown that $F_n = A_t \cdots A_1 P_n$ where

$$A_q = I_r \otimes B_L \qquad L = 2^q,\ r = n/L$$

with

$$B_L = \begin{bmatrix} I_{L/2} & \Omega_{L/2} \\ I_{L/2} & -\Omega_{L/2} \end{bmatrix} \quad \text{and} \quad \Omega_{L/2} = \text{diag}(1, \omega_L, \ldots, \omega_L^{L/2-1}).$$

The matrix P_n is called the *bit reversal permutation*, the description of which we omit. (Recall the definition of the Kronecker product "$\otimes$" from §4.5.5.) Note that with this factorization, $y = F_n x$ can be computed as follows:

$x = P_n x$
for $q = 1{:}t$
  $L = 2^q,\ r = n/L$ (4.6.5)
  $x = (I_r \otimes B_L)x$
end

The matrices $A_q = (I_r \otimes B_L)$ have 2 nonzeros per row and it is this sparsity that makes it possible to implement the DFT in $O(n \log n)$ flops. In fact, a careful implementation involves $5n \log_2 n$ flops.

The DFT matrix has the property that

$$F_n^{-1} = \frac{1}{n} F_n^H = \frac{1}{n} \bar{F}_n. \qquad (4.6.6)$$

That is, the inverse of F_n is obtained by conjugating its entries and scaling by n. A fast inverse DFT can be obtained from a (forward) FFT merely by replacing all root-of-unity references with their complex conjugate and scaling by n at the end.

The value of the DFT is that many "hard problems" are made simple by transforming into Fourier space (via F_n). The sought-after solution is then obtained by transforming the Fourier space solution into original coordinates (via F_n^{-1}).

Problems

P4.6.1 Show that if $V = V(x_0, \ldots, x_n)$, then

$$\det(V) = \prod_{n \geq i > j \geq 0} (x_i - x_j).$$

P4.6.2 (Gautschi 1975a) Verify the following inequality for the $n = 1$ case above:

$$\| V^{-1} \|_\infty \leq \max_{0 \leq k \leq n} \prod_{\substack{i=0 \\ i \neq k}}^{n} \frac{1 + |x_i|}{|x_k - x_i|}.$$

Equality results if the x_i are all on the same ray in the complex plane.

P4.6.3 Suppose $w = \left[1,\ \omega_n,\ \omega_n^2, \ldots,\ \omega_n^{n/2-1}\right]$ where $n = 2^t$. Using colon notation, express

$$\left[1,\ \omega_r,\ \omega_r^2, \ldots,\ \omega_r^{r/2-1}\right]$$

as a subvector of w where $r = 2^q$, $q = 1{:}t$.

P4.6.4 Prove (4.6.6).

P4.6.5 Expand the operation $x = (I \otimes B_L)x$ in (4.6.5) into a double loop and count the number of flops required by your implementation. (Ignore the details of $x = P_n x$.

P4.6.6 Suppose $n = 3m$ and examine

$$G = [\, F_n(:,0{:}3{:}n-1)\ \ F_n(1{:}3{:}n-1)\ \ F_n(:,2{:}3{:}n-1)\,]$$

as a 3-by-3 block matrix, looking for scaled copies of F_m. Based on what you find, develop a recursive radix-3 FFT analogous to the radix-2 implementation in the text.

Notes and References for Sec. 4.6

Our discussion of Vandermonde linear systems is drawn from the papers

A. Björck and V. Pereyra (1970). "Solution of Vandermonde Systems of Equations," *Math. Comp.* *24*, 893–903.

A. Björck and T. Elfving (1973). "Algorithms for Confluent Vandermonde Systems," *Numer. Math.* *21*, 130–37.

The divided difference computations we discussed are detailed in chapter 2 of

S.D. Conte and C. de Boor (1980). *Elementary Numerical Analysis: An Algorithmic Approach*, 3rd ed., McGraw-Hill, New York.

The latter reference includes an Algol procedure. Error analyses of Vandermonde system solvers include

N.J. Higham (1987b). "Error Analysis of the Björck-Pereyra Algorithms for Solving Vandermonde Systems," *Numer. Math.* *50*, 613–632.

N.J. Higham (1988a). "Fast Solution of Vandermonde-like Systems Involving Orthogonal Polynomials," *IMA J. Num. Anal.* *8*, 473–486.

N.J. Higham (1990). "Stability Analysis of Algorithms for Solving Confluent Vandermonde-like Systems," *SIAM J. Matrix Anal. Appl.* *11*, 23–41.

S.G. Bartels and D.J. Higham (1992). "The Structured Sensitivity of Vandermonde-Like Systems," *Numer. Math.* *62*, 17–34.

J.M. Varah (1993). "Errors and Perturbations in Vandermonde Systems," *IMA J. Num. Anal.* *13*, 1–12.

Interesting theoretical results concerning the condition of Vandermonde systems may be found in

W. Gautschi (1975a). "Norm Estimates for Inverses of Vandermonde Matrices," *Numer. Math.* *23*, 337–47.

W. Gautschi (1975b). "Optimally Conditioned Vandermonde Matrices," *Numer. Math.* *24*, 1–12.

The basic algorithms presented can be extended to cover confluent Vandermonde systems, block Vandermonde systems, and Vandermonde systems that are based on other polynomial bases:

G. Galimberti and V. Pereyra (1970). "Numerical Differentiation and the Solution of Multidimensional Vandermonde Systems," *Math. Comp.* *24*, 357–64.

G. Galimberti and V. Pereyra (1971). "Solving Confluent Vandermonde Systems of Hermitian Type," *Numer. Math.* *18*, 44–60.

H. Van de Vel (1977). "Numerical Treatment of a Generalized Vandermonde systems of Equations," *Lin. Alg. and Its Applic.* *17*, 149–74.

G.H. Golub and W.P Tang (1981). "The Block Decomposition of a Vandermonde Matrix and Its Applications," *BIT* *21*, 505–17.

D. Calvetti and L. Reichel (1992). "A Chebychev-Vandermonde Solver," *Lin. Alg. and Its Applic.* *172*, 219–229.

D. Calvetti and L. Reichel (1993). "Fast Inversion of Vandermonde-Like Matrices Involving Orthogonal Polynomials," *BIT* *33*, 473–484.

H. Lu (1994). "Fast Solution of Confluent Vandermonde Linear Systems," *SIAM J. Matrix Anal. Appl.* *15*, 1277–1289.

H. Lu (1996). "Solution of Vandermonde-like Systems and Confluent Vandermonde-like Systems," *SIAM J. Matrix Anal. Appl.* *17*, 127–138.

The FFT literature is very extensive and scattered. For an overview of the area couched in Kronecker product notation, see

C.F. Van Loan (1992). *Computational Frameworks for the Fast Fourier Transform*, SIAM Publications, Philadelphia, PA.

The point of view in this text is that different FFTs correspond to different factorizations of the DFT matrix. These are *sparse* factorizations in that the factors have very few nonzeros per row.

4.7 Toeplitz and Related Systems

Matrices whose entries are constant along each diagonal arise in many applications and are called *Toeplitz matrices*. Formally, $T \in \mathbb{R}^{n\times n}$ is Toeplitz if there exist scalars $r_{-n+1}, \ldots, r_0, \ldots, r_{n-1}$ such that $a_{ij} = r_{j-i}$ for all i and j. Thus,

$$T = \begin{bmatrix} r_0 & r_1 & r_2 & r_3 \\ r_{-1} & r_0 & r_1 & r_2 \\ r_{-2} & r_{-1} & r_0 & r_1 \\ r_{-3} & r_{-2} & r_{-1} & r_0 \end{bmatrix} = \begin{bmatrix} 3 & 1 & 7 & 6 \\ 4 & 3 & 1 & 7 \\ 0 & 4 & 3 & 1 \\ 9 & 0 & 4 & 3 \end{bmatrix}$$

is Toeplitz.

Toeplitz matrices belong to the larger class of *persymmetric matrices*. We say that $B \in \mathbb{R}^{n\times n}$ is persymmetric if it symmetric about its northeast-southwest diagonal, i.e., $b_{ij} = b_{n-j+1,n-i+1}$ for all i and j. This is equivalent to requiring $B = EB^TE$ where $E = [\,e_n, \ldots, e_1\,] = I_n(:, n: -1:1)$ is the n-by-n *exchange matrix*, i.e.,

$$E = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

It is easy to verify that (a) Toeplitz matrices are persymmetric and (b) the inverse of a nonsingular Toeplitz matrix is persymmetric. In this section we show how the careful exploitation of (b) enables us to solve Toeplitz systems with $O(n^2)$ flops. The discussion focuses on the important case when T is also symmetric and positive definite. Unsymmetric Toeplitz systems and connections with circulant matrices and the discrete Fourier transform are briefly discussed.

4.7.1 Three Problems

Assume that we have scalars $r_1, \ldots, r_n$ such that for $k = 1{:}n$ the matrices

$$T_k = \begin{bmatrix} 1 & r_1 & \cdots & r_{k-2} & r_{k-1} \\ r_1 & 1 & \ddots & & r_{k-2} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ r_{k-2} & & \ddots & \ddots & r_1 \\ r_{k-1} & r_{k-2} & \cdots & r_1 & 1 \end{bmatrix}$$

are positive definite. (There is no loss of generality in normalizing the diagonal.) Three algorithms are described in this section:

- Durbin's algorithm for the *Yule-Walker problem* $T_ny = -[r_1, \ldots, r_n]^T$.
- Levinson's algorithm for the general righthand side problem $T_nx = b$.
- Trench's algorithm for computing $B = T_n^{-1}$.

In deriving these methods, we denote the k-by-k exchange matrix by E_k, i.e., $E_k = I_k(:, k: -1:1)$.

4.7.2 Solving the Yule-Walker Equations

We begin by presenting Durbin's algorithm for the Yule-Walker equations which arise in conjunction with certain linear prediction problems. Suppose for some k that satisfies $1 \le k \le n-1$ we have solved the k-th order Yule-Walker system $T_ky = -r = -(r_1, \ldots, r_k)^T$. We now show how the $(k+1)$-st order Yule-Walker system

$$\begin{bmatrix} T_k & E_kr \\ r^TE_k & 1 \end{bmatrix} \begin{bmatrix} z \\ \alpha \end{bmatrix} = -\begin{bmatrix} r \\ r_{k+1} \end{bmatrix}$$

can be solved in $O(k)$ flops. First observe that

$$z = T_k^{-1}(-r - \alpha E_kr) = y - \alpha T_k^{-1}E_kr$$

and

$$\alpha = -r_{k+1} - r^TE_kz.$$

Since T_k^{-1} is persymmetric, $T_k^{-1}E_k = E_kT_k^{-1}$ and thus,

$$z = y - \alpha E_kT_k^{-1}r = y + \alpha E_ky.$$

By substituting this into the above expression for α we find

$$\alpha = -r_{k+1} - r^TE_k(y + \alpha E_ky) = -(r_{k+1} + r^TE_ky)/(1 + r^Ty).$$

The denominator is positive because T_{k+1} is positive definite and because

$$\begin{bmatrix} I & E_ky \\ 0 & 1 \end{bmatrix}^T \begin{bmatrix} T_k & E_kr \\ r^TE_k & 1 \end{bmatrix} \begin{bmatrix} I & E_ky \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} T_k & 0 \\ 0 & 1 + r^Ty \end{bmatrix}.$$

We have illustrated the kth step of an algorithm proposed by Durbin (1960). It proceeds by solving the Yule-Walker systems

$$T_ky^{(k)} = -r^{(k)} = -[r_1, \ldots, r_k]^T$$

for $k = 1{:}n$ as follows:

$y^{(1)} = -r_1$
for $k = 1{:}n-1$
    $\beta_k = 1 + [r^{(k)}]^T y^{(k)}$
    $\alpha_k = -(r_{k+1} + r^{(k)^T} E_k y^{(k)})/\beta_k$ (4.7.1)
    $z^{(k)} = y^{(k)} + \alpha_k E_k y^{(k)}$
    $y^{(k+1)} = \begin{bmatrix} z^{(k)} \\ \alpha_k \end{bmatrix}$
end

As it stands, this algorithm would require $3n^2$ flops to generate $y = y^{(n)}$. It is possible, however, to reduce the amount of work even further by exploiting some of the above expressions:

$$\begin{aligned}
\beta_k &= 1 + [r^{(k)}]^T y^{(k)} \\
&= 1 + \begin{bmatrix} r^{(k-1)^T} & r_k \end{bmatrix} \begin{bmatrix} y^{(k-1)} + \alpha_{k-1} E_{k-1} y^{(k-1)} \\ \alpha_{k-1} \end{bmatrix} \\
&= (1 + [r^{(k-1)}]^T y^{(k-1)}) \; + \; \alpha_{k-1}\left([r^{(k-1)}]^T E_{k-1} y^{(k-1)} + r_k\right) \\
&= \beta_{k-1} + \alpha_{k-1}(-\beta_{k-1}\alpha_{k-1}) \\
&= (1 - \alpha_{k-1}^2)\beta_{k-1}.
\end{aligned}$$

Using this recursion we obtain the following algorithm:

Algorithm 4.7.1. (Durbin) Given real numbers $1 = r_0, r_1, \ldots, r_n$ such that $T = (r_{|i-j|}) \in \mathbb{R}^{n \times n}$ is positive definite, the following algorithm computes $y \in \mathbb{R}^n$ such that $Ty = -(r_1, \ldots, r_n)^T$.

$y(1) = -r(1)$; $\beta = 1$; $\alpha = -r(1)$
for $k = 1{:}n-1$
    $\beta = (1 - \alpha^2)\beta$
    $\alpha = -\left(r(k+1) + r(k{:}-1{:}1)^T y(1{:}k)\right)/\beta$
    $z(1{:}k) = y(1{:}k) + \alpha y(k{:}-1{:}1)$
    $y(1{:}k+1) = \begin{bmatrix} z(1{:}k) \\ \alpha \end{bmatrix}$
end

This algorithm requires $2n^2$ flops. We have included an auxiliary vector z for clarity, but it can be avoided.

Example 4.7.1 Suppose we wish to solve the Yule-Walker system

$$\begin{bmatrix} 1 & .5 & .2 \\ .5 & 1 & .5 \\ .2 & .5 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = -\begin{bmatrix} .5 \\ .2 \\ .1 \end{bmatrix}$$

using Algorithm 4.7.1. After one pass through the loop we obtain

$$\alpha = 1/15, \qquad \beta = 3/4, \qquad y = \begin{bmatrix} -8/15 \\ 1/15 \end{bmatrix}.$$

We then compute

$$\begin{aligned}
\beta &= (1-\alpha^2)\beta = 56/75 \\
\alpha &= -(r_3 + r_2 y_1 + r_1 y_2)/\beta = -1/28 \\
z_1 &= y_1 + \alpha y_2 = -225/420 \\
z_2 &= y_2 + \alpha y_1 = -36/420,
\end{aligned}$$

giving the final solution $y = [-75,\ 12,\ -5]^T/140$.

4.7.3 The General Right Hand Side Problem

With a little extra work, it is possible to solve a symmetric positive definite Toeplitz system that has an arbitrary right-hand side. Suppose that we have solved the system

$$T_k x = b = (b_1, \ldots, b_k)^T \tag{4.7.2}$$

for some k satisfying $1 \le k < n$ and that we now wish to solve

$$\begin{bmatrix} T_k & E_k r \\ r^T E_k & 1 \end{bmatrix} \begin{bmatrix} v \\ \mu \end{bmatrix} = \begin{bmatrix} b \\ b_{k+1} \end{bmatrix}. \tag{4.7.3}$$

Here, $r = (r_1, \ldots, r_k)^T$ as above. Assume also that the solution to the kth order Yule-Walker system $T_k y = -r$ is also available. From $T_k v + \mu E_k r = b$ it follows that

$$v = T_k^{-1}(b - \mu E_k r) = x - \mu T_k^{-1} E_k = x + \mu E_k y$$

and so

$$\begin{aligned}
\mu &= b_{k+1} - r^T E_k v \\
&= b_{k+1} - r^T E_k x - \mu r^T y \\
&= \left(b_{k+1} - r^T E_k x\right) / \left(1 + r^T y\right).
\end{aligned}$$

Consequently, we can effect the transition from (4.7.2) to (4.7.3) in $O(k)$ flops.

Overall, we can efficiently solve the system $T_n x = b$ by solving the systems $T_k x^{(k)} = b^{(k)} = (b_1, \ldots, b_k)^T$ and $T_k y^{(k)} = -r^{(k)} = (r_1, \ldots, r_k)^T$ "in parallel" for $k = 1{:}n$. This is the gist of the following algorithm:

Algorithm 4.7.2 (Levinson) Given $b \in \mathbb{R}^n$ and real numbers $1 = r_0, r_1, \ldots, r_n$ such that $T = (r_{|i-j|}) \in \mathbb{R}^{n \times n}$ is positive definite, the following algorithm computes $x \in \mathbb{R}^n$ such that $Tx = b$.

$y(1) = -r(1)$; $x(1) = b(1)$; $\beta = 1$; $\alpha = -r(1)$

for $k = 1{:}n-1$
  $\beta = (1-\alpha^2)\beta$; $\mu = \left(b(k+1) - r(1{:}k)^T x(k{:}-1{:}1)\right)/\beta$
  $v(1{:}k) = x(1{:}k) + \mu y(k{:}-1{:}1)$
  $x(1{:}k+1) = \begin{bmatrix} v(1{:}k) \\ \mu \end{bmatrix}$
  if $k < n-1$
    $\alpha = \left(-r(k+1) + r(1{:}k)^T y(k{:}-1{:}1)\right)/\beta$
    $z(1{:}k) = y(1{:}k) + \alpha y(k{:}-1{:}1)$
    $y(1{:}k+1) = \begin{bmatrix} z(1{:}k) \\ \alpha \end{bmatrix}$
  end
end

This algorithm requires $4n^2$ flops. The vectors z and v are for clarity and can be avoided in a detailed implementation.

Example 4.7.2 Suppose we wish to solve the symmetric positive definite Toeplitz system

$$\begin{bmatrix} 1 & .5 & .2 \\ .5 & 1 & .5 \\ .2 & .5 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = -\begin{bmatrix} 4 \\ -1 \\ 3 \end{bmatrix}$$

using the above algorithm. After one pass through the loop we obtain

$$\alpha = 1/15, \qquad \beta = 3/4, \qquad y = \begin{bmatrix} -8/15 \\ 1/15 \end{bmatrix} \qquad x = \begin{bmatrix} 6 \\ -4 \end{bmatrix}.$$

We then compute

$$\begin{aligned} \beta &= (1-\alpha^2)\beta = 56/75 \qquad & \mu &= (b_3 - r_1 x_2 - r_2 x_1)/\beta = 285/56 \\ v_1 &= x_1 + \mu y_2 = 355/56 \qquad & v_2 &= x_2 + \mu y_1 = -376/56 \end{aligned}$$

giving the final solution $x = [355,\ -376,\ 285]^T/56$.

4.7.4 Computing the Inverse

One of the most surprising properties of a symmetric positive definite Toeplitz matrix T_n is that its complete inverse can be calculated in $O(n^2)$ flops. To derive the algorithm for doing this, partition T_n^{-1} as follows

$$T_n^{-1} = \begin{bmatrix} A & Er \\ r^T E & 1 \end{bmatrix}^{-1} = \begin{bmatrix} B & v \\ v^T & \gamma \end{bmatrix} \tag{4.7.4}$$

where $A = T_{n-1}$, $E = E_{n-1}$, and $r = (r_1, \ldots, r_{n-1})^T$. From the equation

$$\begin{bmatrix} A & Er \\ r^T E & 1 \end{bmatrix} \begin{bmatrix} v \\ \gamma \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

it follows that $Av = -\gamma Er = -\gamma E(r_1, \ldots, r_{n-1})^T$ and $\gamma = 1 - r^T Ev$. If y solves the $(n-1)$-st order Yule-Walker system $Ay = -r$, then these

expressions imply that

$$\begin{aligned} \gamma &= 1/(1 + r^T y) \\ v &= \gamma E y . \end{aligned}$$

Thus, the last row and column of T_n^{-1} are readily obtained.

It remains for us to develop working formulae for the entries of the submatrix B in (4.7.4). Since $AB + Erv^T = I_{n-1}$, it follows that

$$B = A^{-1} - (A^{-1}Er)v^T = A^{-1} + \frac{vv^T}{\gamma}.$$

Now since $A = T_{n-1}$ is nonsingular and Toeplitz, its inverse is persymmetric. Thus,

$$\begin{aligned} b_{ij} &= (A^{-1})_{ij} + \frac{v_i v_j}{\gamma} \\ &= (A^{-1})_{n-j,n-i} + \frac{v_i v_j}{\gamma} \\ &= b_{n-j,n-i} - \frac{v_{n-j}v_{n-i}}{\gamma} + \frac{v_i v_j}{\gamma} \\ &= b_{n-j,n-i} + \frac{1}{\gamma}\left(v_i v_j - v_{n-j}v_{n-i}\right) . \end{aligned} \tag{4.7.5}$$

This indicates that although B is not persymmetric, we can readily compute an element b_{ij} from its reflection across the northeast-southwest axis. Coupling this with the fact that A^{-1} is persymmetric enables us to determine B from its "edges" to its "interior."

Because the order of operations is rather cumbersome to describe, we preview the formal specification of the algorithm pictorially. To this end, assume that we know the last column and row of T_n^{-1}:

$$T_n^{-1} = \begin{bmatrix} u & u & u & u & u & k \\ u & u & u & u & u & k \\ u & u & u & u & u & k \\ u & u & u & u & u & k \\ u & u & u & u & u & k \\ k & k & k & k & k & k \end{bmatrix}.$$

Here u and k denote the unknown and the known entries respectively, and $n = 6$. Alternately exploiting the persymmetry of T_n^{-1} and the recursion (4.7.5), we can compute B, the leading $(n-1)$-by-$(n-1)$ block of T_{n-1}^{-1}, as follows:

$$\xrightarrow{\text{persym.}} \begin{bmatrix} k & k & k & k & k & k \\ k & u & u & u & u & k \\ k & u & u & u & u & k \\ k & u & u & u & u & k \\ k & u & u & u & u & k \\ k & k & k & k & k & k \end{bmatrix} \xrightarrow{(4.7.5)} \begin{bmatrix} k & k & k & k & k & k \\ k & u & u & u & k & k \\ k & u & u & u & k & k \\ k & u & u & u & k & k \\ k & k & k & k & k & k \\ k & k & k & k & k & k \end{bmatrix}$$

$$\overset{\text{persym.}}{\longrightarrow}\begin{bmatrix} k & k & k & k & k & k \\ k & k & k & k & k & k \\ k & k & u & u & k & k \\ k & k & u & u & k & k \\ k & k & k & k & k & k \\ k & k & k & k & k & k \end{bmatrix} \overset{(4.7.5)}{\longrightarrow} \begin{bmatrix} k & k & k & k & k & k \\ k & k & k & k & k & k \\ k & k & u & k & k & k \\ k & k & k & k & k & k \\ k & k & k & k & k & k \\ k & k & k & k & k & k \end{bmatrix}$$

$$\overset{\text{persym.}}{\longrightarrow}\begin{bmatrix} k & k & k & k & k & k \\ k & k & k & k & k & k \\ k & k & k & k & k & k \\ k & k & k & k & k & k \\ k & k & k & k & k & k \\ k & k & k & k & k & k \end{bmatrix}.$$

Of course, when computing a matrix that is both symmetric and persymmetric, such as T_n^{-1}, it is only necessary to compute the "upper wedge" of the matrix—e.g.,

$$\begin{matrix} \times & \times & \times & \times & \times & \times \\ & \times & \times & \times & \times & \\ & & \times & \times & & \end{matrix} \qquad (n = 6)$$

With this last observation, we are ready to present the overall algorithm.

Algorithm 4.7.3 (Trench) Given real numbers $1 = r_0, r_1, \ldots, r_n$ such that $T = (r_{|i-j|}) \in \mathbb{R}^{n \times n}$ is positive definite, the following algorithm computes $B = T_n^{-1}$. Only those b_{ij} for which $i \leq j$ and $i + j \leq n + 1$ are computed.

```
Use Algorithm 4.7.1 to solve T_{n-1}y = -(r_1,...,r_{n-1})^T.
γ = 1/(1 + r(1:n-1)^T y(1:n-1))
v(1:n-1) = γy(n-1:-1:1)
B(1,1) = γ
B(1,2:n) = v(n-1:-1:1)^T
for i = 2:floor((n-1)/2) + 1
    for j = i:n-i+1
        B(i,j) = B(i-1,j-1)+
                (v(n+1-j)v(n+1-i) - v(i-1)v(j-1)) /γ
    end
end
```

This algorithm requires $13n^2/4$ flops.

Example 4.7.3 If the above algorithm is applied to compute the inverse B of the positive definite Toeplitz matrix

$$\begin{bmatrix} 1 & .5 & .2 \\ .5 & 1 & .5 \\ .2 & .5 & 1 \end{bmatrix},$$

then we obtain $\gamma = 75/56$, $b_{11} = 75/56$, $b_{12} = -5/7$, $b_{13} = 5/56$, and $b_{22} = 12/7$.

4.7.5 Stability Issues

Error analyses for the above algorithms have been performed by Cybenko (1978), and we briefly report on some of his findings.

The key quantities turn out to be the α_k in (4.7.1). In exact arithmetic these scalars satisfy

$$|\alpha_k| < 1$$

and can be used to bound $\| T^{-1} \|_1$:

$$\max\left\{ \frac{1}{\prod_{j=1}^{n-1}(1-\alpha_j^2)}, \frac{1}{\prod_{j=1}^{n-1}(1-\alpha_j)} \right\} \leq \| T_n^{-1} \| \leq \prod_{j=1}^{n-1} \frac{1+|\alpha_j|}{1-|\alpha_j|} \quad (4.7.6)$$

Moreover, the solution to the Yule-Walker system $T_n y = -r(1{:}n)$ satisfies

$$\| y \|_1 = \left(\prod_{k=1}^{n} (1 + \alpha_k) \right) - 1 \qquad (4.7.7)$$

provided all the α_k are non-negative.

Now if $\hat{x}$ is the computed Durbin solution to the Yule-Walker equations then $r_D = T_n\hat{x} + r$ can be bounded as follows

$$\| r_D \| \approx \mathbf{u} \prod_{k=1}^{n} (1 + |\hat{\alpha}_k|)$$

where $\hat{\alpha}_k$ is the computed version of α_k. By way of comparison, since each $|r_i|$ is bounded by unity, it follows that $\| r_C \| \approx \mathbf{u} \| y \|_1$ where r_C is the residual associated with the computed solution obtained via Cholesky. Note that the two residuals are of comparable magnitude provided (4.7.7) holds. Experimental evidence suggests that this is the case even if some of the α_k are negative. Similar comments apply to the numerical behavior of the Levinson algorithm.

For the Trench method, the computed inverse $\hat{B}$ of T_n^{-1} can be shown to satisfy

$$\frac{\| T_n^{-1} - \hat{B} \|_1}{\| T_n^{-1} \|_1} \approx \mathbf{u} \prod_{k=1}^{n} \frac{1 + |\hat{\alpha}_k|}{1 - |\hat{\alpha}_k|}.$$

In light of (4.7.7) we see that the right-hand side is an approximate upper bound for $\mathbf{u}\| T_n^{-1} \|$ which is approximately the size of the relative error when T_n^{-1} is calculated using the Cholesky factorization.

4.7.6 The Unsymmetric Case

Similar recursions can be developed for the unsymmetric case. Suppose we are given scalars $r_1, \ldots, r_{n-1}, p_1, \ldots, p_{n-1}$, and $b_1, \ldots, b_n$ and that we want to solve a linear system $Tx = b$ of the form

$$\begin{bmatrix} 1 & r_1 & r_2 & r_3 & r_4 \\ p_1 & 1 & r_1 & r_2 & r_3 \\ p_2 & p_1 & 1 & r_1 & r_2 \\ p_3 & p_2 & p_1 & 1 & r_1 \\ p_4 & p_3 & p_2 & p_1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix} \qquad (n = 5).$$

In the process that follows we require the leading principle submatrices $T_k = T(1{:}k, 1{:}k)$, $k = 1{:}n$ to be nonsingular. Using the same notation as above, it can shown that if we have the solutions to the k-by-k systems

$$\begin{aligned} T_k^T y &= -r &&= -[r_1\, r_2 \cdots r_k]^T \\ T_k w &= -p &&= -[p_1\, p_2 \cdots p_k]^T \\ T_k x &= b &&= [b_1\, b_2 \cdots b_k]^T, \end{aligned} \qquad (4.7.8)$$

then we can obtain solutions to

$$\begin{aligned} \begin{bmatrix} T_k & E_k r \\ p^T E_k & 1 \end{bmatrix}^T \begin{bmatrix} z \\ \alpha \end{bmatrix} &= -\begin{bmatrix} r \\ r_{k+1} \end{bmatrix} \\ \begin{bmatrix} T_k & E_k r \\ p^T E_k & 1 \end{bmatrix} \begin{bmatrix} u \\ \nu \end{bmatrix} &= -\begin{bmatrix} p \\ p_{k+1} \end{bmatrix} \\ \begin{bmatrix} T_k & E_k r \\ p^T E_k & 1 \end{bmatrix} \begin{bmatrix} v \\ \mu \end{bmatrix} &= \begin{bmatrix} b \\ b_{k+1} \end{bmatrix} \end{aligned} \qquad (4.7.9)$$

in $O(k)$ flops. This means that in principle it is possible to solve an unsymmetric Toeplitz system in $O(n^2)$ flops. However, the stability of the process cannot be assured unless the matrices $T_k = T(1{:}k, 1{:}k)$ are sufficiently well conditioned.

4.7.7 Circulant Systems

A very important class of Toeplitz matrices are the *circulant* matrices. Here is an example:

$$C(v) = \begin{bmatrix} v_0 & v_4 & v_3 & v_2 & v_1 \\ v_1 & v_0 & v_4 & v_3 & v_2 \\ v_2 & v_1 & v_0 & v_4 & v_3 \\ v_3 & v_2 & v_1 & v_0 & v_4 \\ v_4 & v_3 & v_2 & v_1 & v_0 \end{bmatrix}.$$

Notice that each column of a circulant is a "downshifted" version of its predecessor. In particular, if we define the downshift permutation S_n by

$$S_n = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \qquad (n = 5)$$

and $v = [v_0\, v_1 \cdots v_{n-1}]^T$, then $C(v) = [\, v,\ S_n v,\ S_n^2 v, \ldots,\ S_n^{n-1} v\,]$.

There are important connections between circulant matrices, Toeplitz matrices, and the DFT. First of all, it can be shown that

$$C(v) = F_n^{-1} \text{diag}(F_n v) F_n. \qquad (4.7.10)$$

This means that a product of the form $y = C(v)x$ can be solved at "FFT speed":

$$\begin{aligned} \tilde{x} &= F_n x \\ \tilde{v} &= F_n v \\ z &= \tilde{v} .* \tilde{x} \\ y &= F_n^{-1} z \end{aligned}$$

In other words, three DFTs and a vector multiply suffice to carry out the product of a circulant matrix and a vector. Products of this form are called *convolutions* and they are ubiquitous in signal processing and other areas.

Toeplitz-vector products can also be computed fast. The key idea is that any Toeplitz matrix can be "embedded" in a circulant. For example,

$$T = \begin{bmatrix} 5 & 2 & 7 \\ 4 & 5 & 2 \\ 9 & 4 & 5 \end{bmatrix}$$

is the leading 3-by-3 submatrix of

$$C = \left[\begin{array}{ccc|cc} 5 & 2 & 7 & 9 & 4 \\ 4 & 5 & 2 & 7 & 9 \\ 9 & 4 & 5 & 2 & 7 \\ \hline 7 & 9 & 4 & 5 & 2 \\ 2 & 7 & 9 & 4 & 5 \end{array}\right].$$

In general, if $T = (t_{ij})$ is an n-by-n Toeplitz matrix, then $T = C(1{:}n, 1{:}n)$ where $C \in \mathbb{R}^{(2n-1)\times(2n-1)}$ is a circulant with

$$C(:,1) = \begin{bmatrix} T(1{:}n, 1) \\ T(1, n{:}-1{:}2)^T \end{bmatrix}.$$

Note that if $y = Cx$ and $x(n+1{:}2n-1) = 0$, then $y(1{:}n) = Tx(1{:}n)$ showing that Toeplitz vector products can also be computed at "FFT speed."

Problems

P4.7.1 For any $v \in \mathbb{R}^n$ define the vectors $v_+ = (v + E_n v)/2$ and $v_- = (v - E_n v)/2$. Suppose $A \in \mathbb{R}^{n\times n}$ is symmetric and persymmetric. Show that if $Ax = b$ then $Ax_+ = b_+$ and $Ax_- = b_-$.

P4.7.2 Let $U \in \mathbb{R}^{n\times n}$ be the unit upper triangular matrix with the property that: $U(1{:}k-1, k) = E_{k-1}y^{(k-1)}$ where $y^{(k)}$ is defined by (4.7.1). Show that

$$U^T T_n U = \text{diag}(1, \beta_1, \ldots, \beta_{n-1}).$$

P4.7.3 Suppose $z \in \mathbb{R}^n$ and that $S \in \mathbb{R}^{n\times n}$ is orthogonal. Show that if

$$X = \left[z,\ Sz,\ \ldots, S^{n-1}z\right]$$

then $X^T X$ is Toeplitz.

P4.7.4 Consider the LDL^T factorization of an n-by-n symmetric, tridiagonal, positive definite Toeplitz matrix. Show that d_n and $\ell_{n,n-1}$ converge as $n \to \infty$.

P4.7.5 Show that the product of two lower triangular Toeplitz matrices is Toeplitz.

P4.7.6 Give an algorithm for determining $\mu \in \mathbb{R}$ such that

$$T_n + \mu\left(e_n e_1^T + e_1 e_n^T\right)$$

is singular. Assume $T_n = (r_{|i-j|})$ is positive definite, with $r_0 = 1$.

P4.7.7 Rewrite Algorithm 4.7.2 so that it does not require the vectors z and v.

P4.7.8 Give an algorithm for computing $\kappa_\infty(T_k)$ for $k = 1{:}n$.

P4.7.9 Suppose A_1, A_2, A_3 and A_4 are m-by-m matrices and that

$$A = \begin{bmatrix} A_0 & A_1 & A_2 & A_3 \\ A_3 & A_0 & A_1 & A_2 \\ A_2 & A_3 & A_0 & A_1 \\ A_1 & A_2 & A_3 & A_0 \end{bmatrix}.$$

Show that there is a permutation matrix Π such that $\Pi^T A \Pi = C = (C_{ij})$ where each C_{ij} is a 4-by-4 circulant matrix.

P4.7.10 A p-by-p block matrix $A = (A_{ij})$ with m-by-m blocks is *block Toeplitz* if there exist $A_{-p+1}, \ldots, A_{-1}, A_0, A_1, \ldots, A_{p-1} \in \mathbb{R}^{m\times m}$ so that $A_{ij} = A_{i-j}$, e.g.,

$$A = \begin{bmatrix} A_0 & A_1 & A_2 & A_3 \\ A_{-1} & A_0 & A_1 & A_2 \\ A_{-2} & A_{-1} & A_0 & A_1 \\ A_{-3} & A_{-2} & A_{-1} & A_0 \end{bmatrix}.$$

(a) Show that there is a permutation Π such that

$$\Pi^T A \Pi = \begin{bmatrix} T_{11} & T_{12} & \cdots & T_{1m} \\ T_{21} & T_{22} & & \vdots \\ \vdots & & \ddots & \\ T_{m1} & \cdots & & T_{mm} \end{bmatrix}$$

where each T_{ij} is p-by-p and Toeplitz. Each T_{ij} should be "made up" of (i,j) entries selected from the A_k matrices. (b) What can you say about the T_{ij} if $A_k = A_{-k}$, $k = 1{:}p-1$?

P4.7.11 Show how to compute the solutions to the systems in (4.7.9) given that the solutions to the systems in (4.7.8) are available. Assume that all the matrices involved are nonsingular. Proceed to develop a *fast* unsymmetric Toeplitz solver for $Tx = b$ assuming that T's leading principle submatrices are all nonsingular.

P4.7.12 A matrix $H \in \mathbb{R}^{n\times n}$ is *Hankel* if $H(n{:}-1{:}1, :)$ is Toeplitz. Show that if $A \in \mathbb{R}^{n\times n}$ is defined by

$$a_{ij} = \int_0^\pi \cos(k\theta)\cos(j\theta)\,d\theta$$

then A is the sum of a Hankel matrix and Toeplitz matrix. Hint. Make use of the identity $\cos(u+v) = \cos(u)\cos(v) - \sin(u)\sin(v)$.

P4.7.13 Verify that $F_n C(v) = \text{diag}(F_n v)F_n$.

P4.7.14 Show that it is possible to embed a symmetric Toeplitz matrix into a symmetric circulant matrix.

P4.7.15 Consider the kth order Yule-Walker system $T_k y^{(k)} = -r^{(k)}$ that arises in (4.7.1):

$$T_k \begin{bmatrix} y_{k1} \\ \vdots \\ y_{kk} \end{bmatrix} = -\begin{bmatrix} r_1 \\ \vdots \\ r_k \end{bmatrix}.$$

Show that if

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ y_{11} & 1 & 0 & 0 & \cdots & 0 \\ y_{22} & y_{21} & 1 & 0 & \cdots & 0 \\ y_{33} & y_{32} & y_{31} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ y_{n-1,n-1} & y_{n-1,n-2} & y_{n-1,n-3} & \cdots & y_{n-1,1} & 1 \end{bmatrix},$$

then $LT_nL^T = \text{diag}(1, \beta_1, \ldots, \beta_{n-1})$ where $\beta_k = 1 + {r^{(k)}}^T y^{(k)}$. Thus, the Durbin algorithm can be thought of as a fast method for computing the LDL^T factorization of T_n^{-1}.

Notes and References for Sec. 4.7

Anyone who ventures into the vast Toeplitz method literature should first read

J.R. Bunch (1985). "Stability of Methods for Solving Toeplitz Systems of Equations," *SIAM J. Sci. Stat. Comp. 6*, 349–364

for a clarification of stability issues. As is true with the "fast algorithms" area in general, unstable Toeplitz techniques abound and caution must be exercised. See also

G. Cybenko (1978). "Error Analysis of Some Signal Processing Algorithms," Ph.D. thesis, Princeton University.
G. Cybenko (1980). "The Numerical Stability of the Levinson-Durbin Algorithm for Toeplitz Systems of Equations," *SIAM J. Sci. and Stat. Comp. 1*, 303-19.
E. Linzer (1992). "On the Stability of Solution Methods for Band Toeplitz Systems," *Lin.Alg. and Its Application 170*, 1–32.
J.M. Varah (1994). "Backward Error Estimates for Toeplitz Systems," *SIAM J. Matrix Anal. Appl. 15*, 408–417.
A.W. Bojanczyk, R.P. Brent, F.R. de Hoog, and D.R. Sweet (1995). "On the Stability of the Bareiss and Related Toeplitz Factorization Algorithms," *SIAM J. Matrix Anal. Appl. 16*, 40–57.

M. Stewart and P. Van Dooren (1996). "Stability Issues in the Factorization of Structured Matrices," *SIAM J. Matrix Anal. Appl. 18*, to appear.

The original references for the three algorithms described in this section are:

J. Durbin (1960). "The Fitting of Time Series Models," *Rev. Inst. Int. Stat. 28* 233–43.
N. Levinson (1947). "The Weiner RMS Error Criterion in Filter Design and Prediction," *J. Math. Phys. 25*, 261–78.
W.F. Trench (1964). "An Algorithm for the Inversion of Finite Toeplitz Matrices," *J. SIAM 12*, 515–22.

A more detailed description of the nonsymmetric Trench algorithm is given in

S. Zohar (1969). "Toeplitz Matrix Inversion: The Algorithm of W.F. Trench," *J. ACM 16*, 592–601.

Fast Toeplitz system solving has attracted an enormous amount of attention and a sampling of interesting algorithmic ideas may be found in

G. Ammar and W.B. Gragg (1988). "Superfast Solution of Real Positive Definite Toeplitz Systems," *SIAM J. Matrix Anal. Appl. 9*, 61–76.
T.F. Chan and P. Hansen (1992). "A Look-Ahead Levinson Algorithm for Indefinite Toeplitz Systems," *SIAM J. Matrix Anal. Appl. 13*, 490–506.
D.R. Sweet (1993). "The Use of Pivoting to Improve the Numerical Performance of Algorithms for Toeplitz Matrices," *SIAM J. Matrix Anal. Appl. 14*, 468–493.
T. Kailath and J. Chun (1994). "Generalized Displacement Structure for Block-Toeplitz, Toeplitz-Block, and Toeplitz-Derived Matrices," *SIAM J. Matrix Anal. Appl. 15*, 114–128.
T. Kailath and A.H. Sayed (1995). "Displacement Structure: Theory and Applications," *SIAM Review 37*, 297–386.

Important Toeplitz matrix applications are discussed in

J. Makhoul (1975). "Linear Prediction: A Tutorial Review," *Proc. IEEE* 63(4), 561–80.
J. Markel and A. Gray (1976). *Linear Prediction of Speech*, Springer-Verlag, Berlin and New York.
A.V. Oppenheim (1978). *Applications of Digital Signal Processing* , Prentice-Hall, Englewood Cliffs.

Hankel matrices are constant along their antidiagonals and arise in several important areas.

G. Heinig and P. Jankowski (1990). "Parallel and Superfast Algorithms for Hankel Systems of Equations," *Numer. Math. 58*, 109–127.
R.W. Freund and H. Zha (1993). "A Look-Ahead Algorithm for the Solution of General Hankel Systems," *Numer. Math. 64*, 295–322.

The DFT/Toeplitz/circulant connection is discussed in

C.F. Van Loan (1992). *Computational Frameworks for the Fast Fourier Transform*, SIAM Publications, Philadelphia, PA.

Chapter 5

Orthogonalization and Least Squares

§5.1 Householder and Givens Matrices
§5.2 The QR Factorization
§5.3 The Full Rank LS Problem
§5.4 Other Orthogonal Factorizations
§5.5 The Rank Deficient LS Problem
§5.6 Weighting and Iterative Improvement
§5.7 Square and Underdetermined Systems

This chapter is primarily concerned with the least squares solution of overdetermined systems of equations, i.e., the minimization of $\| Ax - b \|_2$ where $A \in \mathbb{R}^{m\times n}$ with $m \geq n$ and $b \in \mathbb{R}^m$. The most reliable solution procedures for this problem involve the reduction of A to various canonical forms via orthogonal transformations. Householder reflections and Givens rotations are central to this process and we begin the chapter with a discussion of these important transformations. In §5.2 we discuss the computation of the factorization $A = QR$ where Q is orthogonal and R is upper triangular. This amounts to finding an orthonormal basis for ran(A). The QR factorization can be used to solve the full rank least squares problem as we show in §5.3. The technique is compared with the method of normal equations after a perturbation theory is developed. In §5.4 and §5.5 we consider methods for handling the difficult situation when A is rank deficient (or nearly so). QR with column pivoting and the SVD are featured. In §5.6 we discuss several steps that can be taken to improve the quality of a computed

M. Stewart and P. Van Dooren (1996). "Stability Issues in the Factorization of Structured Matrices," *SIAM J. Matrix Anal. Appl. 18*, to appear.

The original references for the three algorithms described in this section are:

J. Durbin (1960). "The Fitting of Time Series Models," *Rev. Inst. Int. Stat. 28* 233–43.
N. Levinson (1947). "The Weiner RMS Error Criterion in Filter Design and Prediction," *J. Math. Phys. 25*, 261–78.
W.F. Trench (1964). "An Algorithm for the Inversion of Finite Toeplitz Matrices," *J. SIAM 12*, 515–22.

A more detailed description of the nonsymmetric Trench algorithm is given in

S. Zohar (1969). "Toeplitz Matrix Inversion: The Algorithm of W.F. Trench," *J. ACM 16*, 592–601.

Fast Toeplitz system solving has attracted an enormous amount of attention and a sampling of interesting algorithmic ideas may be found in

G. Ammar and W.B. Gragg (1988). "Superfast Solution of Real Positive Definite Toeplitz Systems," *SIAM J. Matrix Anal. Appl. 9*, 61–76.
T.F. Chan and P. Hansen (1992). "A Look-Ahead Levinson Algorithm for Indefinite Toeplitz Systems," *SIAM J. Matrix Anal. Appl. 13*, 490–506.
D.R. Sweet (1993). "The Use of Pivoting to Improve the Numerical Performance of Algorithms for Toeplitz Matrices," *SIAM J. Matrix Anal. Appl. 14*, 468–493.
T. Kailath and J. Chun (1994). "Generalized Displacement Structure for Block-Toeplitz, Toeplitz-Block, and Toeplitz-Derived Matrices," *SIAM J. Matrix Anal. Appl. 15*, 114–128.
T. Kailath and A.H. Sayed (1995). "Displacement Structure: Theory and Applications," *SIAM Review 37*, 297–386.

Important Toeplitz matrix applications are discussed in

J. Makhoul (1975). "Linear Prediction: A Tutorial Review," *Proc. IEEE* 63(4), 561–80.
J. Markel and A. Gray (1976). *Linear Prediction of Speech*, Springer-Verlag, Berlin and New York.
A.V. Oppenheim (1978). *Applications of Digital Signal Processing* , Prentice-Hall, Englewood Cliffs.

Hankel matrices are constant along their antidiagonals and arise in several important areas.

G. Heinig and P. Jankowski (1990). "Parallel and Superfast Algorithms for Hankel Systems of Equations," *Numer. Math. 58*, 109–127.
R.W. Freund and H. Zha (1993). "A Look-Ahead Algorithm for the Solution of General Hankel Systems," *Numer. Math. 64*, 295–322.

The DFT/Toeplitz/circulant connection is discussed in

C.F. Van Loan (1992). *Computational Frameworks for the Fast Fourier Transform*, SIAM Publications, Philadelphia, PA.

Chapter 5

Orthogonalization and Least Squares

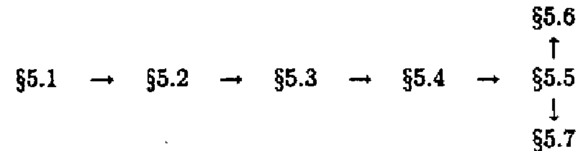
§5.1 Householder and Givens Matrices
§5.2 The QR Factorization
§5.3 The Full Rank LS Problem
§5.4 Other Orthogonal Factorizations
§5.5 The Rank Deficient LS Problem
§5.6 Weighting and Iterative Improvement
§5.7 Square and Underdetermined Systems

This chapter is primarily concerned with the least squares solution of overdetermined systems of equations, i.e., the minimization of $\| Ax - b \|_2$ where $A \in \mathbb{R}^{m\times n}$ with $m \geq n$ and $b \in \mathbb{R}^m$. The most reliable solution procedures for this problem involve the reduction of A to various canonical forms via orthogonal transformations. Householder reflections and Givens rotations are central to this process and we begin the chapter with a discussion of these important transformations. In §5.2 we discuss the computation of the factorization $A = QR$ where Q is orthogonal and R is upper triangular. This amounts to finding an orthonormal basis for ran(A). The QR factorization can be used to solve the full rank least squares problem as we show in §5.3. The technique is compared with the method of normal equations after a perturbation theory is developed. In §5.4 and §5.5 we consider methods for handling the difficult situation when A is rank deficient (or nearly so). QR with column pivoting and the SVD are featured. In §5.6 we discuss several steps that can be taken to improve the quality of a computed

least squares solution. Some remarks about square and underdetermined systems are offered in §5.7.

Before You Begin

Chapters 1, 2, and 3 and §§4.1-4.3 are assumed. Within this chapter there are the following dependencies:



Complementary references include Lawson and Hanson (1974), Farebrother (1987), and Björck (1996). See also Stewart (1973), , Hager (1988), Stewart and Sun (1990), Watkins (1991), Gill, Murray, and Wright (1991), Higham (1996), Trefethen and Bau (1996), and Demmel (1996). Some MATLAB functions important to this chapter are qr, svd, pinv, orth, rank, and the "backslash" operator "\." LAPACK connections include

LAPACK: Householder/Givens Tools	
_LARFG	Generates a Householder matrix
_LARF	Householder times matrix
_LARFX	Small n Householder times matrix
_LARFB	Block Householder times matrix
_LARFT	Computes $I - VTV^H$ block reflector representation
_LARTG	Generates a plane rotation
_LARGV	Generates a vector of plane rotations
_LARTV	Applies a vector of plane rotations to a vector pair
_LASR	Applies rotation sequence to a matrix
CSROT	Real rotation times complex vector pair
CROT	Complex rotation (c real) times complex vector pair
CLACGV	Complex rotation (s real) times complex vector pair

LAPACK: Orthogonal Factorizations	
_GEQRF	$A = QR$
_GEQPF	$A\Pi = QR$
_ORMQR	Q (factored form) times matrix (real case)
_UNMQR	Q (factored form) times matrix (complex case)
_ORGQR	Generates Q (real case)
_UNGQR	Generates Q (complex case)
_GERQF	$A = RQ$ = (upper triangular)(orthogonal)
_GELQF	$A = QL$ = (orthogonal)(lower triangular)
_GEQLF	$A = LQ$ = (lower triangular)(orthogonal)
_TZRQF	$A = RQ$ where A is upper trapezoidal
_GESVD	$A = U\Sigma V^T$
_BDSQR	SVD of real bidiagonal matrix
_GEBRD	Bidiagonalization of general matrix
_ORGBR	Generates the orthogonal transformations
_GBBRD	Bidiagonalization of band matrix

LAPACK: Least Squares	
_GELS	Full rank min $\| AX - B \|_{2F}$ or min $\| A^H X - B \|_F$
_GELSS	SVD solution to min $\| AX - B \|_F$
_GELSX	Complete orthogonal decomposition solution to min $\| AX - B \|_F$.
_GEEQU	Equilibrates general matrix to reduce condition

5.1 Householder and Givens Matrices

Recall that $Q \in \mathbb{R}^{n\times n}$ is *orthogonal* if $Q^TQ = QQ^T = I_n$. Orthogonal matrices have an important role to play in least squares and eigenvalue computations. In this section we introduce the key players in this game: Householder reflections and Givens rotations.

5.1.1 A 2-by-2 Preview

It is instructive to examine the geometry associated with rotations and reflections at the $n = 2$ level. A 2-by-2 orthogonal matrix Q is a *rotation* if it has the form

$$Q = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}.$$

If $y = Q^Tx$, then y is obtained by rotating x counterclockwise through an angle θ.

A 2-by-2 orthogonal matrix Q is a *reflection* if it has the form

$$Q = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ \sin(\theta) & -\cos(\theta) \end{bmatrix}.$$

If $y = Q^Tx = Qx$, then y is obtained by reflecting the vector x across the line defined by

$$S = \text{span}\left\{\begin{bmatrix} \cos(\theta/2) \\ \sin(\theta/2) \end{bmatrix}\right\}.$$

Reflections and rotations are computationally attractive because they are easily constructed and because they can be used to introduce zeros in a vector by properly choosing the rotation angle or the reflection plane.

Example 5.1.1 Suppose $x = [\,1,\ \sqrt{3}\,]^T$. If we set

$$Q = \begin{bmatrix} \cos(-60^\circ) & \sin(-60^\circ) \\ -\sin(-60^\circ) & \cos(-60^\circ) \end{bmatrix} = \begin{bmatrix} 1/2 & -\sqrt{3}/2 \\ \sqrt{3}/2 & 1/2 \end{bmatrix}$$

then $Q^Tx = [\,2,\ 0\,]^T$. Thus, a rotation of -60° zeros the second component of x. If

$$Q = \begin{bmatrix} \cos(30^\circ) & \sin(30^\circ) \\ \sin(30^\circ) & -\cos(30^\circ) \end{bmatrix} = \begin{bmatrix} \sqrt{3}/2 & 1/2 \\ 1/2 & -\sqrt{3}/2 \end{bmatrix}$$

then $Q^Tx = [\,2,\ 0\,]^T$. Thus, by reflecting x across the 30° line we can zero its second component.

5.1.2 Householder Reflections

Let $v \in \mathbb{R}^n$ be nonzero. An n-by-n matrix P of the form

$$P = I - \frac{2}{v^Tv}vv^T \tag{5.1.1}$$

is called a *Householder reflection.* (Synonyms: Householder matrix, Householder transformation.) The vector v is called a *Householder vector.* If a vector x is multiplied by P, then it is reflected in the hyperplane $\text{span}\{v\}^\perp$. It is easy to verify that Householder matrices are symmetric and orthogonal.

Householder reflections are similar in two ways to Gauss transformations, which we introduced in §3.2.1. They are rank-1 modifications of the identity and they can be used to zero selected components of a vector. In particular, suppose we are given $0 \neq x \in \mathbb{R}^n$ and want Px to be a multiple of $e_1 = I_n(:,1)$. Note that

$$Px = \left(I - \frac{2vv^T}{v^Tv}\right)x = x - \frac{2v^Tx}{v^Tv}v$$

and $Px \in \text{span}\{e_1\}$ imply $v \in \text{span}\{x, e_1\}$. Setting $v = x + \alpha e_1$ gives

$$v^Tx = x^Tx + \alpha x_1$$

and

$$v^Tv = x^Tx + 2\alpha x_1 + \alpha^2,$$

and therefore

$$Px = \left(1 - 2\frac{x^Tx + \alpha x_1}{x^Tx + 2\alpha x_1 + \alpha^2}\right)x - 2\alpha\frac{v^Tx}{v^Tv}e_1 .$$

In order for the coefficient of x to be zero, we set $\alpha = \pm\| x \|_2$ for then

$$v = x \pm \| x \|_2 e_1 \;\Rightarrow\; Px = \left(I - 2\frac{vv^T}{v^Tv}\right)x = \mp\| x \|_2 e_1. \tag{5.1.2}$$

It is this simple determination of v that makes the Householder reflection so useful.

Example 5.1.2 If $x = [3,\ 1,\ 5,\ 1]^T$ and $v = [9,\ 1,\ 5,\ 1]^T$, then

$$P = I - 2\frac{vv^T}{v^Tv} = \frac{1}{54}\begin{bmatrix} -27 & -9 & -45 & -9 \\ -9 & 53 & -5 & -1 \\ -45 & -5 & 29 & -5 \\ -9 & -1 & -5 & 53 \end{bmatrix}$$

has the property that $Px = [-6,\ 0,\ 0,\ 0,\ ]^T$.

5.1.3 Computing the Householder Vector

There are a number of important practical details associated with the determination of a Householder matrix, i.e., the determination of a Householder vector. One concerns the choice of sign in the definition of v in (5.1.2). Setting

$$v_1 = x_1 - \| x \|_2$$

has the nice property that Px is a positive multiple of e_1. But this recipe is dangerous if x is close to a positive multiple of e_1 because severe cancellation would occur. However, the formula

$$v_1 = x_1 - \| x \|_2 = \frac{x_1^2 - \| x \|_2^2}{x_1 + \| x \|_2} = \frac{-(x_2^2 + \cdots + x_n^2)}{x_1 + \| x \|_2}$$

suggested by Parlett (1971) does not suffer from this defect in the $x_1 > 0$ case.

In practice, it is handy to normalize the Householder vector so that $v(1) = 1$. This permits the storage of $v(2{:}n)$ where the zeros have been introduced in x, i.e., $x(2{:}n)$. We refer to $v(2{:}n)$ as the *essential part* of the Householder vector. Recalling that $\beta = 2/v^Tv$ and letting **length**(x) specify vector dimension, we obtain the following encapsulation:

Algorithm 5.1.1 (Householder Vector) Given $x \in \mathbb{R}^n$, this function computes $v \in \mathbb{R}^n$ with $v(1) = 1$ and $\beta \in \mathbb{R}$ such that $P = I_n - \beta vv^T$ is orthogonal and $Px = \| x \|_2 e_1$.

```
function: [v, β] = house(x)
    n = length(x)
    σ = x(2:n)^T x(2:n)
    v = [ 1 ; x(2:n) ]
    if σ = 0
        β = 0
    else
        μ = sqrt(x(1)^2 + σ)
        if x(1) <= 0
            v(1) = x(1) - μ
        else
            v(1) = -σ/(x(1) + μ)
        end
        β = 2v(1)^2/(σ + v(1)^2)
        v = v/v(1)
    end
```

This algorithm involves about $3n$ flops and renders a computed Householder matrix that is orthogonal to machine precision, a concept discussed below.

A production version of Algorithm 5.1.1 may involve a preliminary scaling of the x vector ($x \leftarrow x/\| x \|$) to avoid overflow.

5.1.4 Applying Householder Matrices

It is critical to exploit structure when applying a Householder reflection to a matrix. If $A \in \mathbb{R}^{m\times n}$ and $P = I - \beta vv^T \in \mathbb{R}^{m\times m}$, then

$$PA = \left(I - \beta vv^T\right) A = A - vw^T$$

where $w = \beta A^T v$. Likewise, if $P = I - \beta vv^T \in \mathbb{R}^{n\times n}$, then

$$AP = A\left(I - \beta vv^T\right) = A - wv^T$$

where $w = \beta Av$. Thus, an m-by-n Householder update involves a matrix-vector multiplication and an outer product update. It requires $4mn$ flops. Failure to recognize this and to treat P as a general matrix increases work by an order of magnitude. *Householder updates never entail the explicit formation of the Householder matrix.*

Both of the above Householder updates can be implemented in a way that exploits the fact that $v(1) = 1$. This feature can be important in the computation of PA when m is small and in the computation of AP when n is small.

As an example of a Householder matrix update, suppose we want to overwrite $A \in \mathbb{R}^{m\times n}$ ($m \geq n$) with $B = Q^T A$ where Q is an orthogonal matrix chosen so that $B(j+1{:}m, j) = 0$ for some j that satisfies $1 \leq j \leq n$. In addition, suppose $A(j{:}m, 1{:}j-1) = 0$ and that we want to store the essential part of the Householder vector in $A(j+1{:}m, j)$. The following instructions accomplish this task:

$$\begin{aligned}
&[v,\beta] = \textbf{house}(A(j{:}m,j))\\
&A(j{:}m,j{:}n) = (I_{m-j+1} - \beta vv^T)A(j{:}m,j{:}n)\\
&A(j+1{:}m,j) = v(2{:}m-j+1)
\end{aligned}$$

From the computational point of view, we have applied an order $m-j+1$ Householder matrix to the bottom $m-j+1$ rows of A. However, mathematically we have also applied the m-by-m Householder matrix

$$\tilde{P} = \begin{bmatrix} I_{j-1} & 0 \\ 0 & P \end{bmatrix} = I_m - \beta\tilde{v}\tilde{v}^T \qquad \tilde{v} = \begin{bmatrix} 0 \\ v \end{bmatrix}$$

to A in its entirety. Regardless, the "essential" part of the Householder vector can be recorded in the zeroed portion of A.

5.1.5 Roundoff Properties

The roundoff properties associated with Householder matrices are very favorable. Wilkinson (1965, pp. 152-62) shows that **house** produces a Householder vector $\hat{v}$ very near the exact v. If $\hat{P} = I - 2\hat{v}\hat{v}^T/\hat{v}^T\hat{v}$ then

$$\| \hat{P} - P \|_2 = O(\mathbf{u})$$

meaning that $\hat{P}$ is *orthogonal to machine precision*. Moreover, the computed updates with $\hat{P}$ are close to the exact updates with P :

$$\begin{aligned}
fl(\hat{P}A) &= P(A+E) \qquad \| E \|_2 = O(\mathbf{u}\| A \|_2)\\
fl(A\hat{P}) &= (A+E)P \qquad \| E \|_2 = O(\mathbf{u}\| A \|_2)
\end{aligned}$$

5.1.6 Factored Form Representation

Many Householder based factorization algorithms that are presented in the following sections compute products of Householder matrices

$$Q = Q_1 Q_2 \cdots Q_r \qquad Q_j = I - \beta_j v^{(j)} v^{(j)T} \tag{5.1.3}$$

where $r \leq n$ and each $v^{(j)}$ has the form

$$v^{(j)} = (\underbrace{0, 0, \cdots 0,}_{j-1}\ 1\ v^{(j)}_{j+1}, \cdots, v^{(j)}_n)^T .$$

It is usually not necessary to compute Q explicitly even if it is involved in subsequent calculations. For example, if $C \in \mathbb{R}^{n\times q}$ and we wish to compute $Q^T C$, then we merely execute the loop

$$\begin{aligned}
&\textbf{for } j = 1{:}r\\
&\qquad C = Q_j C\\
&\textbf{end}
\end{aligned}$$

The storage of the Householder vectors $v^{(1)} \cdots v^{(r)}$ and the corresponding β_j (if convenient) amounts to a *factored form* representation of Q. To illustrate the economies of the factored form representation, suppose that we have an array A and that $A(j+1{:}n, j)$ houses $v^{(j)}(j+1{:}n)$, the essential part of the jth Householder vector. The overwriting of $C \in \mathbb{R}^{n\times q}$ with $Q^T C$ can then be implemented as follows:

$$\begin{aligned}
&\textbf{for } j = 1{:}r\\
&\qquad v(j{:}n) = \begin{bmatrix} 1 \\ A(j+1{:}n, j) \end{bmatrix} \qquad\qquad (5.1.4)\\
&\qquad C(j{:}n,:) = (I - \beta_j v(j{:}n)v(j{:}n)^T)C(j{:}n,:)\\
&\textbf{end}
\end{aligned}$$

This involves about $2qr(2n - r)$ flops. If Q is explicitly represented as an n-by-n matrix, Q^TC would involve $2n^2q$ flops.

Of course, in some applications, it is necessary to explicitly form Q (or parts of it). Two possible algorithms for computing the Householder product matrix Q in (5.1.3) are *forward accumulation*,

```
Q = I_n
for j = 1:r
    Q = QQ_j
end
```

and *backward accumulation*,

```
Q = I_n
for j = r: - 1:1
    Q = Q_jQ
end
```

Recall that the leading $(j-1)$-by-$(j-1)$ portion of Q_j is the identity. Thus, at the beginning of backward accumulation, Q is "mostly the identity" and it gradually becomes full as the iteration progresses. This pattern can be exploited to reduce the number of required flops. In contrast, Q is full in forward accumulation after the first step. For this reason, backward accumulation is cheaper and the strategy of choice:

$$\begin{array}{l} Q = I_n \\ \textbf{for } j = r: -1:1 \\ \quad v(j{:}n) = \begin{bmatrix} 1 \\ A(j+1{:}n, j) \end{bmatrix} \\ \quad Q(j{:}n, j{:}n) = (I - \beta_j v(j{:}n)v(j{:}n)^T)Q(j{:}n, j{:}n) \\ \textbf{end} \end{array} \tag{5.1.5}$$

This involves about $4(n^2r - nr^2 + r^3/3)$ flops.

5.1.7 A Block Representation

Suppose $Q = Q_1 \cdots Q_r$ is a product of n-by-n Householder matrices as in (5.1.3). Since each Q_j is a rank-one modification of the identity, it follows from the structure of the Householder vectors that Q is a rank-r modification of the identity and can be written in the form

$$Q = I + WY^T \tag{5.1.6}$$

where W and Y are n-by-r matrices. The key to computing the *block representation* (5.1.6) is the following lemma.

Lemma 5.1.1 *Suppose $Q = I + WY^T$ is an n-by-n orthogonal matrix with $W, Y \in \mathbb{R}^{n\times j}$. If $P = I - \beta vv^T$ with $v \in \mathbb{R}^n$ and $z = -\beta Qv$, then*

$$Q_+ = QP = I + W_+Y_+^T$$

where $W_+ = [\,W\ z\,]$ and $Y_+ = [\,Y\ v\,]$ are each n-by-$(j+1)$.

Proof.

$$\begin{aligned} QP &= (I + WY^T)(I - \beta vv^T) = I + WY^T - \beta Qvv^T \\ &= I + WY^T + zv^T = I + [\,W\ z\,][\,Y\ v\,]^T \ \square \end{aligned}$$

By repeatedly applying the lemma, we can generate the block representation of Q in (5.1.3) from the factored form representation as follows:

Algorithm 5.1.2 Suppose $Q = Q_1 \cdots Q_r$ is a product of n-by-n Householder matrices as described in (5.1.3). This algorithm computes matrices $W, Y \in \mathbb{R}^{n\times r}$ such that $Q = I + WY^T$.

```
Y = v^(1)
W = -β_1 v^(1)
for j = 2:r
    z = -β_j(I + WY^T)v^(j)
    W = [W z]
    Y = [Y v^(j)]
end
```

This algorithm involves about $2r^2n - 2r^3/3$ flops if the zeros in the $v^{(j)}$ are exploited. Note that Y is merely the matrix of Householder vectors and is therefore unit lower triangular. Clearly, the central task in the generation of the WY representation (5.1.6) is the computation of the W matrix.

The block representation for products of Householder matrices is attractive in situations where Q must be applied to a matrix. Suppose $C \in \mathbb{R}^{n\times q}$. It follows that the operation

$$C \leftarrow Q^TC = (I + WY^T)^TC = C + Y(W^TC)$$

is rich in level-3 operations. On the other hand, if Q is in factored form, Q^TC is just rich in the level-2 operations of matrix-vector multiplication and outer product updates. Of course, in this context the distinction between level-2 and level-3 diminishes as C gets narrower.

We mention that the "WY" representation is not a generalized Householder transformation from the geometric point of view. True block reflectors have the form $Q = I - 2VV^T$ where $V \in \mathbb{R}^{n\times r}$ satisfies $V^TV = I_r$. See Schreiber and Parlett (1987) and also Schreiber and Van Loan (1989).

Example 5.1.3 If $n = 4$, $r = 2$, and $[\,1,\ .6,\ 0,\ .8\,]^T$ and $[\,0,\ 1,\ .8,\ .6\,]^T$ are the

Householder vectors associated with Q_1 and Q_2 respectively, then

$$Q_1Q_2 = I_4 + WY^T \equiv I_4 + \begin{bmatrix} -1 & 1.080 \\ -.6 & -.352 \\ 0 & -.800 \\ -.8 & .264 \end{bmatrix} \begin{bmatrix} 1 & .6 & 0 & .8 \\ 0 & 1 & .8 & .6 \end{bmatrix}.$$

5.1.8 Givens Rotations

Householder reflections are exceedingly useful for introducing zeros on a grand scale, e.g., the annihilation of all but the first component of a vector. However, in calculations where it is necessary to zero elements more selectively, *Givens rotations* are the transformation of choice. These are rank-two corrections to the identity of the form

$$G(i,k,\theta) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \begin{matrix} \\ \\ i \\ \\ k \\ \\ \\ \end{matrix} \qquad (5.1.7)$$

$$\qquad\qquad i \qquad k$$

where $c = \cos(\theta)$ and $s = \sin(\theta)$ for some θ. Givens rotations are clearly orthogonal.

Premultiplication by $G(i,k,\theta)^T$ amounts to a counterclockwise rotation of θ radians in the (i,k) coordinate plane. Indeed, if $x \in \mathbb{R}^n$ and $y = G(i,k,\theta)^T x$, then

$$y_j = \begin{cases} cx_i - sx_k & j = i \\ sx_i + cx_k & j = k \\ x_j & j \neq i,k \end{cases}.$$

From these formulae it is clear that we can force y_k to be zero by setting

$$c = \frac{x_i}{\sqrt{x_i^2 + x_k^2}} \qquad s = \frac{-x_k}{\sqrt{x_i^2 + x_k^2}} \qquad (5.1.8)$$

Thus, it is a simple matter to zero a specified entry in a vector by using a Givens rotation. In practice, there are better ways to compute c and s than (5.1.8). The following algorithm, for example, guards against overflow.

Algorithm 5.1.3 Given scalars a and b, this function computes $c = \cos(\theta)$ and $s = \sin(\theta)$ so

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}.$$

```
function: [c, s] = givens(a, b)
    if b = 0
        c = 1; s = 0
    else
        if |b| > |a|
            τ = -a/b; s = 1/√(1 + τ²); c = sτ
        else
            τ = -b/a; c = 1/√(1 + τ²); s = cτ
        end
    end
```

This algorithm requires 5 flops and a single square root. Note that it does not compute θ and so it does not involve inverse trigonometric functions.

Example 5.1.4 If $x = [1,\ 2,\ 3,\ 4]^T$, $\cos(\theta) = 1/\sqrt{5}$, and $\sin(\theta) = -2/\sqrt{5}$, then $G(2,4,\theta)x = [1,\ \sqrt{20},\ 3,\ 0]^T$.

5.1.9 Applying Givens Rotations

It is critical that the simple structure of a Givens rotation matrix be exploited when it is involved in a matrix multiplication. Suppose $A \in \mathbb{R}^{m \times n}$, $c = \cos(\theta)$, and $s = \sin(\theta)$. If $G(i,k,\theta) \in \mathbb{R}^{m \times m}$, then the update $A \leftarrow G(i,k,\theta)^T A$ effects just two rows of A,

$$A([i,k],:) = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T A([i,k],:)$$

and requires just $6n$ flops:

```
for j = 1:n
    τ1 = A(i, j)
    τ2 = A(k, j)
    A(1, j) = cτ1 - sτ2
    A(2, j) = sτ1 + cτ2
end
```

Likewise, if $G(i,k,\theta) \in \mathbb{R}^{n \times n}$, then the update $A \leftarrow AG(i,k,\theta)$ effects just two columns of A,

$$A(:,[i,k]) = A(:,[i,k]) \begin{bmatrix} c & s \\ -s & c \end{bmatrix}$$

and requires just $6m$ flops:

```
for j = 1:m
    τ1 = A(j, i)
    τ2 = A(j, k)
    A(j, i) = cτ1 - sτ2
    A(j, k) = sτ1 + cτ2
end
```

5.1.10 Roundoff Properties

The numerical properties of Givens rotations are as favorable as those for Householder reflections. In particular, it can be shown that the computed $\hat{c}$ and $\hat{s}$ in **givens** satisfy

$$\begin{aligned} \hat{c} &= c(1+\epsilon_c) & \epsilon_c &= O(\mathbf{u}) \\ \hat{s} &= s(1+\epsilon_s) & \epsilon_s &= O(\mathbf{u}). \end{aligned}$$

If $\hat{c}$ and $\hat{s}$ are subsequently used in a Givens update, then the computed update is the exact update of a nearby matrix:

$$\begin{aligned} fl[\hat{G}(i,k,\theta)^T A] &= G(i,k,\theta)^T(A+E) & \| E \|_2 &\approx \mathbf{u}\| A \|_2 \\ fl[A\hat{G}(i,k,\theta)] &= (A+E)G(i,k,\theta) & \| E \|_2 &\approx \mathbf{u}\| A \|_2. \end{aligned}$$

A detailed error analysis of Givens rotations may be found in Wilkinson (1965, pp. 131-39).

5.1.11 Representing Products of Givens Rotations

Suppose $Q = G_1 \cdots G_t$ is a product of Givens rotations. As we have seen in connection with Householder reflections, it is more economical to keep the orthogonal matrix Q in factored form than to compute explicitly the product of the rotations. Using a technique demonstrated by Stewart (1976), it is possible to do this in a very compact way. The idea is to associate a single floating point number ρ with each rotation. Specifically, if

$$Z = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \qquad c^2 + s^2 = 1$$

then we define the scalar ρ by

```
if c = 0
    ρ = 1
elseif |s| < |c|
    ρ = sign(c)s/2                                  (5.1.9)
else
    ρ = 2sign(s)/c
end
```


Essentially, this amounts to storing $s/2$ if the sine is smaller and $2/c$ if the cosine is smaller. With this encoding, it is possible to reconstruct $\pm Z$ as follows:

```
if ρ = 1
    c = 0; s = 1
elseif |ρ| < 1
    s = 2ρ; c = √(1 - s²)                           (5.1.10)
else
    c = 2/ρ; s = √(1 - c²)
end
```

That $-Z$ may be generated is usually of no consequence for if Z zeros a particular matrix entry, so does $-Z$. The reason for essentially storing the smaller of c and s is that the formula $\sqrt{1-x^2}$ renders poor results if x is near unity. More details may be found in Stewart (1976). Of course, to "reconstruct" $G(i,k,\theta)$ we need i and k in addition to the associated ρ. This usually poses no difficulty as we discuss in §5.2.3.

5.1.12 Error Propagation

We offer some remarks about the propagation of roundoff error in algorithms that involve sequences of Householder/Givens updates. To be precise, suppose $A = A_0 \in \mathbb{R}^{m \times n}$ is given and that matrices $A_1, \ldots, A_p = B$ are generated via the formula

$$A_k = fl(\hat{Q}_k A_{k-1} \hat{Z}_k) \qquad k = 1:p\,.$$

Assume that the above Householder and Givens algorithms are used for both the generation and application of the $\hat{Q}_k$ and $\hat{Z}_k$. Let Q_k and Z_k be the orthogonal matrices that would be produced in the absence of roundoff. It can be shown that

$$B = (Q_p \cdots Q_1)(A+E)(Z_1 \cdots Z_p), \qquad (5.1.11)$$

where $\| E \|_2 \le c\mathbf{u}\| A \|_2$ and c is a constant that depends mildly on n, m, and p. In plain English, B is an exact orthogonal update of a matrix near to A.

5.1.13 Fast Givens Transformations

The ability to introduce zeros in a selective fashion makes Givens rotations an important zeroing tool in certain structured problems. This has led to the development of "fast Givens" procedures. The fast Givens idea amounts to a clever representation of Q when Q is the product of Givens rotations.

In particular, Q is represented by a matrix pair (M, D) where $M^TM = D = \text{diag}(d_i)$ and each d_i is positive. The matrices Q, M, and D are connected through the formula

$$Q = MD^{-1/2} = M\text{diag}(1/\sqrt{d_i}).$$

Note that $(MD^{-1/2})^T(MD^{-1/2}) = D^{-1/2}DD^{-1/2} = I$ and so the matrix $MD^{-1/2}$ is orthogonal. Moreover, if F is an n-by-n matrix with $F^TDF = D_{new}$ diagonal, then $M_{new}^TM_{new} = D_{new}$ where $M_{new} = MF$. Thus, it is possible to update the fast Givens representation (M, D) to obtain (M_{new}, D_{new}). For this idea to be of practical interest, we must show how to give F zeroing capabilities subject to the constraint that it "keeps" D diagonal.

The details are best explained at the 2-by-2 level. Let $x = [x_1\ x_2]^T$ and $D = \text{diag}(d_1, d_2)$ be given and assume that d_1 and d_2 are positive. Define

$$M_1 = \begin{bmatrix} \beta_1 & 1 \\ 1 & \alpha_1 \end{bmatrix} \tag{5.1.12}$$

and observe that

$$M_1^Tx = \begin{bmatrix} \beta_1x_1 + x_2 \\ x_1 + \alpha_1x_2 \end{bmatrix}$$

and

$$M_1^TDM_1 = \begin{bmatrix} d_2 + \beta_1^2d_1 & d_1\beta_1 + d_2\alpha_1 \\ d_1\beta_1 + d_2\alpha_1 & d_1 + \alpha_1^2d_2 \end{bmatrix} \equiv D_1 .$$

If $x_2 \neq 0$, $\alpha_1 = -x_1/x_2$, and $\beta_1 = -\alpha_1d_2/d_1$, then

$$M_1^Tx = \begin{bmatrix} x_2(1+\gamma_1) \\ 0 \end{bmatrix}$$

$$M_1^TDM_1 = \begin{bmatrix} d_2(1+\gamma_1) & 0 \\ 0 & d_1(1+\gamma_1) \end{bmatrix}$$

where $\gamma_1 = -\alpha_1\beta_1 = (d_2/d_1)(x_1/x_2)^2$.

Analogously, if we assume $x_1 \neq 0$ and define M_2 by

$$M_2 = \begin{bmatrix} 1 & \alpha_2 \\ \beta_2 & 1 \end{bmatrix} \tag{5.1.13}$$

where $\alpha_2 = -x_2/x_1$ and $\beta_2 = -(d_1/d_2)\alpha_2$, then

$$M_2^Tx = \begin{bmatrix} x_1(1+\gamma_2) \\ 0 \end{bmatrix}$$

and

$$M_2^TDM_2 = \begin{bmatrix} d_1(1+\gamma_2) & 0 \\ 0 & d_2(1+\gamma_2) \end{bmatrix} \equiv D_2,$$

where $\gamma_2 = -\alpha_2\beta_2 = (d_1/d_2)(x_2/x_1)^2$.

It is easy to show that for either $i = 1$ or 2, the matrix $J = D^{1/2}M_iD_i^{-1/2}$ is orthogonal and that it is designed so that the second component of $J^T(D^{-1/2}x)$ is zero. (J may actually be a reflection and thus it is half-correct to use the popular term "fast Givens.")

Notice that the γ_i satisfy $\gamma_1\gamma_2 = 1$. Thus, we can always select M_i in the above so that the "growth factor" $(1 + \gamma_i)$ is bounded by 2. Matrices of the form

$$M_1 = \begin{bmatrix} \beta_1 & 1 \\ 1 & \alpha_1 \end{bmatrix} \qquad M_2 = \begin{bmatrix} 1 & \alpha_2 \\ \beta_2 & 1 \end{bmatrix}$$

that satisfy $-1 \leq \alpha_i\beta_i \leq 0$ are 2-by-2 *fast Givens transformations*. Notice that premultiplication by a fast Givens transformation involves half the number of multiplies as premultiplication by an "ordinary" Givens transformation. Also, the zeroing is carried out without an explicit square root.

In the n-by-n case, everything "scales up" as with ordinary Givens rotations. The "type 1" transformations have the form

$$F(i,k,\alpha,\beta) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & \beta & \cdots & 1 & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & 1 & \cdots & \alpha & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \begin{matrix} \\ \\ i \\ \\ k \\ \\ \\ \end{matrix} \tag{5.1.14}$$

$$\qquad\qquad\qquad\qquad i \qquad\quad k$$

while the "type 2" transformations are structured as follows:

$$F(i,k,\alpha,\beta) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & 1 & \cdots & \alpha & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & \beta & \cdots & 1 & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \begin{matrix} \\ \\ i \\ \\ k \\ \\ \\ \end{matrix} \tag{5.1.15}$$

$$\qquad\qquad\qquad\qquad i \qquad\quad k$$

Encapsulating all this we obtain

Algorithm 5.1.4 Given $x \in \mathbb{R}^2$ and positive $d \in \mathbb{R}^2$, the following algorithm computes a 2-by-2 fast Givens transformation M such that the

second component of $M^T x$ is zero and $M^T DM = D_1$ is diagonal where $D = \text{diag}(d_1, d_2)$. If *type* = 1 then M has the form (5.1.12) while if *type* = 2 then M has the form (5.1.13). The diagonal elements of D_1 overwrite d.

```
function: [α, β, type] = fast.givens(x, d)
    if x(2) ≠ 0
        α = −x(1)/x(2); β = −αd(2)/d(1); γ = −αβ
        if γ ≤ 1
            type = 1
            τ = d(1); d(1) = (1 + γ)d(2); d(2) = (1 + γ)τ
        else
            type = 2
            α = 1/α; β = 1/β; γ = 1/γ
            d(1) = (1 + γ)d(1); d(2) = (1 + γ)d(2)
        end
    else
        type = 2
        α = 0; β = 0
    end
```

The application of fast Givens transformations is analogous to that for ordinary Givens transformations. Even with the appropriate type of transformation used, the growth factor $1 + \gamma$ may still be as large as two. Thus, 2^s growth can occur in the entries of D and M after s updates. This means that the diagonal D must be monitored during a fast Givens procedure to avoid overflow. See Anda and Park (1994) for how to do this efficiently.

Nevertheless, element growth in M and D is controlled because at all times we have $MD^{-1/2}$ orthogonal. The roundoff properties of a fast givens procedure are what we would expect of a Givens matrix technique. For example, if we computed $\hat{Q} = fl(\hat{M}\hat{D}^{-1/2})$ where $\hat{M}$ and $\hat{D}$ are the computed M and D, then $\hat{Q}$ is orthogonal to working precision: $\| \hat{Q}^T\hat{Q} - I \|_2 \approx \mathbf{u}$.

Problems

P5.1.1 Execute house with $x = [\,1,\ 7,\ 2,\ 3,\ -1\,]^T$.

P5.1.2 Let x and y be nonzero vectors in $\mathbb{R}^n$. Give an algorithm for determining a Householder matrix P such that Px is a multiple of y.

P5.1.3 Suppose $x \in \mathbb{C}^n$ and that $x_1 = |x_1|e^{i\theta}$ with $\theta \in \mathbb{R}$. Assume $x \neq 0$ and define $u = x + e^{i\theta}\| x \|_2 e_1$ Show that $P = I - 2uu^H/u^H u$ is unitary and that $Px = -e^{i\theta}\| x \|_2 e_1$.

P5.1.4 Use Householder matrices to show that $\det(I + xy^T) = 1 + x^T y$ where x and y are given n-vectors.

P5.1.5 Suppose $x \in \mathbb{C}^2$. Give an algorithm for determining a unitary matrix of the form

$$Q = \begin{bmatrix} c & s \\ -\bar{s} & c \end{bmatrix} \qquad c \in \mathbb{R},\ c^2 + |s|^2 = 1$$

such that the second component of $Q^H x$ is zero.

P5.1.6 Suppose x and y are unit vectors in $\mathbb{R}^n$. Give an algorithm using Givens transformations which computes an orthogonal Q such that $Q^T x = y$.

P5.1.7 Determine $c = \cos(\theta)$ and $s = \sin(\theta)$ such that

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} 5 \\ 12 \end{bmatrix} = \begin{bmatrix} 13 \\ 0 \end{bmatrix}.$$

P5.1.8 Suppose that $Q = I + YTY^T$ is orthogonal where $Y \in \mathbb{R}^{n \times j}$ and $T \in \mathbb{R}^{j \times j}$ is upper triangular. Show that if $Q_+ = QP$ where $P = I - 2vv^T/v^Tv$ is a Householder matrix, then Q_+ can be expressed in the form $Q_+ = I + Y_+T_+Y_+^T$ where $Y_+ \in \mathbb{R}^{n \times (j+1)}$ and $T_+ \in \mathbb{R}^{(j+1)\times(j+1)}$ is upper triangular.

P5.1.9 Give a detailed implementation of Algorithm 5.1.2 with the assumption that $v^{(j)}(j+1{:}n)$, the essential part of the the jth Householder vector, is stored in $A(j+1{:}n, j)$. Since Y is effectively represented in A, your procedure need only set up the W matrix.

P5.1.10 Show that if S is skew-symmetric ($S^T = -S$), then $Q = (I + S)(I - S)^{-1}$ is orthogonal. (Q is called the *Cayley transform* of S.) Construct a rank-2 S so that if x is a vector then Qx is zero except in the first component.

P5.1.11 Suppose $P \in \mathbb{R}^{n \times n}$ satisfies $\| P^TP - I_n \|_2 = \epsilon < 1$. Show that all the singular values of P are in the interval $[1-\epsilon, 1+\epsilon]$ and that $\| P - UV^T \|_2 \leq \epsilon$ where $P = U\Sigma V^T$ is the SVD of P.

P5.1.12 Suppose $A \in \mathbb{R}^{2 \times 2}$. Under what conditions is the closest rotation to A closer than the closest reflection to A?

Notes and References for Sec. 5.1

Householder matrices are named after A.S. Householder, who popularized their use in numerical analysis. However, the properties of these matrices have been known for quite some time. See

H.W. Turnbull and A.C. Aitken (1961). *An Introduction to the Theory of Canonical Matrices*, Dover Publications, New York, pp. 102–5.

Other references concerned with Householder transformations include

A.R. Gourlay (1970). "Generalization of Elementary Hermitian Matrices," *Comp. J. 13*, 411–12.

B.N. Parlett (1971). "Analysis of Algorithms for Reflections in Bisectors," *SIAM Review 13*, 197–208.

N.K. Tsao (1975). "A Note on Implementing the Householder Transformations." *SIAM J. Num. Anal. 12*, 53–58.

B. Danloy (1976). "On the Choice of Signs for Householder Matrices," *J. Comp. Appl. Math. 2*, 67–69.

J.J.M. Cuppen (1984). "On Updating Triangular Products of Householder Matrices," *Numer. Math. 45*, 403–410.

L. Kaufman (1987). "The Generalized Householder Transformation and Sparse Matrices," *Lin. Alg. and Its Applic. 90*, 221–234.

A detailed error analysis of Householder transformations is given in Lawson and Hanson (1974, 83–89).

The basic references for block Householder representations and the associated computations include

C.H. Bischof and C. Van Loan (1987). "The WY Representation for Products of Householder Matrices," *SIAM J. Sci. and Stat. Comp. 8*, s2–s13.

R. Schreiber and B.N. Parlett (1987). "Block Reflectors: Theory and Computation," *SIAM J. Numer. Anal.* *25*, 189-205.
B.N. Parlett and R. Schreiber (1988). "Block Reflectors: Theory and Computation," *SIAM J. Num. Anal.* *25*, 189–205.
R.S. Schreiber and C. Van Loan (1989). "A Storage-Efficient WY Representation for Products of Householder Transformations," *SIAM J. Sci. and Stat. Comp.* *10*, 52–57.
C. Puglisi (1992). "Modification of the Householder Method Based on the Compact WY Representation," *SIAM J. Sci. and Stat. Comp.* *13*, 723–726.
X. Sun and C.H. Bischof (1995). "A Basis-Kernel Representation of Orthogonal Matrices," *SIAM J. Matrix Anal. Appl.* *16*, 1184–1196.

Givens rotations, named after W. Givens, are also referred to as Jacobi rotations. Jacobi devised a symmetric eigenvalue algorithm based on these transformations in 1846. See §8.4. The Givens rotation storage scheme discussed in the text is detailed in

G.W. Stewart (1976). "The Economical Storage of Plane Rotations," *Numer. Math.* *25*, 137–38.

Fast Givens transformations are also referred to as "square-root-free" Givens transformations. (Recall that a square root must ordinarily be computed during the formation of Givens transformation.) There are several ways fast Givens calculations can be arranged. See

M. Gentleman (1973). "Least Squares Computations by Givens Transformations without Square Roots," *J. Inst. Math. Appl.* *12*, 329–36.
C.F. Van Loan (1973). "Generalized Singular Values With Algorithms and Applications," Ph.D. thesis, University of Michigan, Ann Arbor.
S. Hammarling (1974). "A Note on Modifications to the Givens Plane Rotation," *J. Inst. Math. Appl.* *13*, 215–18.
J.H. Wilkinson (1977). "Some Recent Advances in Numerical Linear Algebra," in *The State of the Art in Numerical Analysis*, ed. D.A.H. Jacobs, Academic Press, New York, pp. 1–53.
A.A. Anda and H. Park (1994). "Fast Plane Rotations with Dynamic Scaling," *SIAM J. Matrix Anal. Appl.* *15*, 162–174.

5.2 The QR Factorization

We now show how Householder and Givens transformations can be used to compute various factorizations, beginning with the QR factorization. The QR factorization of an m-by-n matrix A is given by

$$A = QR$$

where $Q \in \mathbb{R}^{m\times m}$ is orthogonal and $R \in \mathbb{R}^{m\times n}$ is upper triangular. In this section we assume $m \geq n$. We will see that if A has full column rank, then the first n columns of Q form an orthonormal basis for $\text{ran}(A)$. Thus, calculation of the QR factorization is one way to compute an orthonormal basis for a set of vectors. This computation can be arranged in several ways. We give methods based on Householder, block Householder, Givens, and fast Givens transformations. The Gram-Schmidt orthogonalization process and a numerically more stable variant called modified Gram-Schmidt are also discussed.

5.2.1 Householder QR

We begin with a QR factorization method that utilizes Householder transformations. The essence of the algorithm can be conveyed by a small example. Suppose $m = 6$, $n = 5$, and assume that Householder matrices H_1 and H_2 have been computed so that

$$H_2H_1A = \begin{bmatrix} \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \boxtimes & \times & \times \\ 0 & 0 & \boxtimes & \times & \times \\ 0 & 0 & \boxtimes & \times & \times \\ 0 & 0 & \boxtimes & \times & \times \end{bmatrix}.$$

Concentrating on the highlighted entries, we determine a Householder matrix $\tilde{H}_3 \in \mathbb{R}^{4\times 4}$ such that

$$\tilde{H}_3 \begin{bmatrix} \boxtimes \\ \boxtimes \\ \boxtimes \\ \boxtimes \end{bmatrix} = \begin{bmatrix} \times \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

If $H_3 = \text{diag}(I_2, \tilde{H}_3)$, then

$$H_3H_2H_1A = \begin{bmatrix} \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times & \times \end{bmatrix}.$$

After n such steps we obtain an upper triangular $H_nH_{n-1}\cdots H_1A = R$ and so by setting $Q = H_1\cdots H_n$ we obtain $A = QR$.

Algorithm 5.2.1 (Householder QR) Given $A \in \mathbb{R}^{m\times n}$ with $m \geq n$, the following algorithm finds Householder matrices $H_1, \ldots, H_n$ such that if $Q = H_1\cdots H_n$, then $Q^TA = R$ is upper triangular. The upper triangular part of A is overwritten by the upper triangular part of R and components $j+1{:}m$ of the jth Householder vector are stored in $A(j+1{:}m, j), j < m$.

```
for j = 1:n
    [v, β] = house(A(j:m, j))
    A(j:m, j:n) = (I_{m-j+1} - βvv^T)A(j:m, j:n)
    if j < m
        A(j + 1:m, j) = v(2:m - j + 1)
    end
end
```

This algorithm requires $2n^2(m - n/3)$ flops.

To clarify how A is overwritten, if

$$v^{(j)} = [\underbrace{0,\ldots,0}_{j-1}, 1, v_{j+1}^{(j)}, \ldots, v_m^{(j)}]^T$$

is the jth Householder vector, then upon completion

$$A = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} & r_{15} \\ v_2^{(1)} & r_{22} & r_{23} & r_{24} & r_{25} \\ v_3^{(1)} & v_3^{(2)} & r_{33} & r_{34} & r_{35} \\ v_4^{(1)} & v_4^{(2)} & v_4^{(3)} & r_{44} & r_{45} \\ v_5^{(1)} & v_5^{(2)} & v_5^{(3)} & v_5^{(4)} & r_{55} \\ v_6^{(1)} & v_6^{(2)} & v_6^{(3)} & v_6^{(4)} & v_6^{(5)} \end{bmatrix}.$$

If the matrix $Q = H_1 \cdots H_n$ is required, then it can be accumulated using (5.1.5). This accumulation requires $4(m^2n - mn^2 + n^3/3)$ flops.

The computed upper triangular matrix $\hat{R}$ is the exact R for a nearby A in the sense that $Z^T(A+E) = \hat{R}$ where Z is some exact orthogonal matrix and $\| E \|_2 \approx \mathbf{u}\| A \|_2$.

5.2.2 Block Householder QR Factorization

Algorithm 5.2.1 is rich in the level-2 operations of matrix-vector multiplication and outer product updates. By reorganizing the computation and using the block Householder representation discussed in §5.1.7 we can obtain a level-3 procedure. The idea is to apply clusters of Householder transformations that are represented in the WY form of §5.1.7.

A small example illustrates the main idea. Suppose $n = 12$ and that the "blocking parameter" r has the value $r = 3$. The first step is to generate Householders H_1, H_2, and H_3 as in Algorithm 5.2.1. However, unlike Algorithm 5.2.1 where the H_i are applied to all of A, we only apply H_1, H_2, and H_3 to $A(:,1{:}3)$. After this is accomplished we generate the block representation $H_1H_2H_3 = I + W_1Y_1^T$ and then perform the level-3 update

$$A(:,4{:}12) = (I + WY^T)A(:,4{:}12).$$

Next, we generate H_4, H_5, and H_6 as in Algorithm 5.2.1. However, these transformations are not applied to $A(:,7{:}12)$ until their block representation $H_4H_5H_6 = I + W_2Y_2^T$ is found. This illustrates the general pattern.

$\lambda = 1$; $k = 0$
while $\lambda \le n$
  $\tau = \min(\lambda + r - 1, n)$; $k = k + 1$
  Using Algorithm 5.2.1, upper triangularize $A(\lambda{:}m, \lambda{:}n)$
    generating Householder matrices $H_\lambda, \ldots, H_\tau$. (5.2.1)
  Use Algorithm 5.1.2 to get the block representation
    $I + W_kY_k = H_\lambda, \ldots, H_\tau$.
  $A(\lambda{:}m, \tau+1{:}n) = (I + W_kY_k^T)^T A(\lambda{:}m, \tau+1{:}n)$
  $\lambda = \tau + 1$
end

The zero-nonzero structure of the Householder vectors that define the matrices $H_\lambda, \ldots, H_\tau$ implies that the first $\lambda - 1$ rows of W_k and Y_k are zero. This fact would be exploited in a practical implementation.

The proper way to regard (5.2.1) is through the partitioning

$$A = [\,A_1, \ldots, A_N\,] \qquad N = \text{ceil}(n/r)$$

where block column A_k is processed during the kth step. In the kth step of (5.2.1), a block Householder is formed that zeros the subdiagonal portion of A_k. The remaining block columns are then updated.

The roundoff properties of (5.2.1) are essentially the same as those for Algorithm 5.2.1. There is a slight increase in the number of flops required because of the W-matrix computations. However, as a result of the blocking, all but a small fraction of the flops occur in the context of matrix multiplication. In particular, the level-3 fraction of (5.2.1) is approximately $1 - 2/N$. See Bischof and Van Loan (1987) for further details.

5.2.3 Givens QR Methods

Givens rotations can also be used to compute the QR factorization. The 4-by-3 case illustrates the general idea:

$$\begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix} \xrightarrow{(3,4)} \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ 0 & \times & \times \end{bmatrix} \xrightarrow{(2,3)} \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \end{bmatrix} \xrightarrow{(1,2)}$$

$$\begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \end{bmatrix} \xrightarrow{(3,4)} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \end{bmatrix} \xrightarrow{(2,3)} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ 0 & 0 & \times \end{bmatrix} \xrightarrow{(3,4)} R$$

Here we have highlighted the 2-vectors that define the underlying Givens rotations. Clearly, if G_j denotes the jth Givens rotation in the reduction, then $Q^TA = R$ is upper triangular where $Q = G_1 \cdots G_t$ and t is the total

number of rotations. For general m and n we have:

Algorithm 5.2.2 (Givens QR) Given $A \in \mathbb{R}^{m\times n}$ with $m \geq n$, the following algorithm overwrites A with $Q^TA = R$, where R is upper triangular and Q is orthogonal.

for $j = 1{:}n$
  for $i = m{:} -1{:}j+1$
    $[c, s] = \textbf{givens}(A(i-1,j), A(i,j))$
    $A(i-1{:}i, j{:}n) = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T A(i-1{:}i, j{:}n)$
  end
end

This algorithm requires $3n^2(m-n/3)$ flops. Note that we could use (5.1.9) to encode (c, s) in a single number ρ which could then be stored in the zeroed entry $A(i,j)$. An operation such as $x \leftarrow Q^Tx$ could then be implemented by using (5.1.10), taking care to reconstruct the rotations in the proper order.

Other sequences of rotations can be used to upper triangularize A. For example, if we replace the for statements in Algorithm 5.2.2 with

for $i = m{:} -1{:}2$
  for $j = 1{:}\min\{i-1, n\}$

then the zeros in A are introduced row-by-row.

Another parameter in a Givens QR procedure concerns the planes of rotation that are involved in the zeroing of each a_{ij}. For example, instead of rotating rows $i-1$ and i to zero a_{ij} as in Algorithm 5.2.2, we could use rows j and i:

for $j = 1{:}n$
  for $i = m{:} -1{:}j+1$
    $[c, s] = \textbf{givens}(A(j,j), A(i,j))$
    $A([\,j\ i\,], j{:}n) = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T A([\,j\ i\,], j{:}n)$
  end
end

5.2.4 Hessenberg QR via Givens

As an example of how Givens rotations can be used in structured problems, we show how they can be employed to compute the QR factorization of an upper Hessenberg matrix. A small example illustrates the general idea.

Suppose $n = 6$ and that after two steps we have computed

$$G(2,3,\theta_2)^TG(1,2,\theta_1)^TA = \begin{bmatrix} \times & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times & \times \\ 0 & 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & 0 & \times & \times \end{bmatrix}.$$

We then compute $G(3,4,\theta_3)$ to zero the current (4,3) entry thereby obtaining

$$G(3,4,\theta_3)^TG(2,3,\theta_2)^TG(1,2,\theta_1)^TA = \begin{bmatrix} \times & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times & \times \\ 0 & 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & 0 & \times & \times \end{bmatrix}.$$

Overall we have

Algorithm 5.2.3 (Hessenberg QR) If $A \in \mathbb{R}^{n\times n}$ is upper Hessenberg, then the following algorithm overwrites A with $Q^TA = R$ where Q is orthogonal and R is upper triangular. $Q = G_1 \cdots G_{n-1}$ is a product of Givens rotations where G_j has the form $G_j = G(j, j+1, \theta_j)$.

for $j = 1{:}n-1$
  $[\,c\ s\,] = \textbf{givens}(A(j,j), A(j+1,j))$
  $A(j{:}j+1, j{:}n) = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T A(j{:}j+1, j{:}n)$
end

This algorithm requires about $3n^2$ flops.

5.2.5 Fast Givens QR

We can use the fast Givens transformations described in §5.1.13 to compute an (M, D) representation of Q. In particular, if M is nonsingular and D is diagonal such that $M^TA = T$ is upper triangular and $M^TM = D$ is diagonal, then $Q = MD^{-1/2}$ is orthogonal and $Q^TA = D^{-1/2}T \equiv R$ is upper triangular. Analogous to the Givens QR procedure we have:

Algorithm 5.2.4 (Fast Givens QR) Given $A \in \mathbb{R}^{m\times n}$ with $m \geq n$, the following algorithm computes nonsingular $M \in \mathbb{R}^{m\times m}$ and positive $d(1{:}m)$ such that $M^TA = T$ is upper triangular, and $M^TM = \text{diag}(d_1, \ldots, d_m)$. A is overwritten by T. Note: $A = (MD^{-1/2})(D^{1/2}T)$ is a QR factorization of A.

```
for i = 1:m
    d(i) = 1
end
for j = 1:n
    for i = m: -1:j+1
        [α, β, type] = fast.givens(A(i-1:i, j), d(i-1:i))
        if type = 1
            A(i-1:i, j:n) = [β 1; 1 α]^T A(i-1:i, j:n)
        else
            A(i-1:i, j:n) = [1 α; β 1]^T A(i-1:i, j:n)
    end
end
```

This algorithm requires $2n^2(m - n/3)$ flops. As we mentioned in the previous section, it is necessary to guard against overflow in fast Givens algorithms such as the above. This means that M, D, and A must be periodically scaled if their entries become large.

If the QR factorization of a narrow band matrix is required, then the fast Givens approach is attractive because it involves no square roots. (We found LDL^T preferable to Cholesky in the narrow band case for the same reason; see §4.3.6.) In particular, if $A \in \mathbb{R}^{m \times n}$ has upper bandwidth q and lower bandwidth p, then $Q^T A = R$ has upper bandwidth $p + q$. In this case Givens QR requires about $O(np(p+q))$ flops and $O(np)$ square roots. Thus, the square roots are a significant portion of the overall computation if $p, q \ll n$.

5.2.6 Properties of the QR Factorization

The above algorithms "prove" that the QR factorization exists. Now we relate the columns of Q to $\text{ran}(A)$ and $\text{ran}(A)^\perp$ and examine the uniqueness question.

Theorem 5.2.1 *If $A = QR$ is a QR factorization of a full column rank $A \in \mathbb{R}^{m \times n}$ and $A = [\,a_1, \ldots, a_n\,]$ and $Q = [\,q_1, \ldots, q_m\,]$ are column partitionings, then*

$$\text{span}\{a_1, \ldots, a_k\} \;=\; \text{span}\{q_1, \ldots, q_k\} \qquad k = 1{:}n\,.$$

In particular, if $Q_1 = Q(1{:}m, 1{:}n)$ and $Q_2 = Q(1{:}m, n+1{:}m)$ then

$$\begin{aligned} \text{ran}(A) &= \text{ran}(Q_1) \\ \text{ran}(A)^\perp &= \text{ran}(Q_2) \end{aligned}$$

and $A = Q_1 R_1$ with $R_1 = R(1{:}n, 1{:}n)$.

Proof. Comparing kth columns in $A = QR$ we conclude that

$$a_k \;=\; \sum_{i=1}^{k} r_{ik} q_i \;\in\; \text{span}\{q_1, \ldots, q_k\}\;. \tag{5.2.2}$$

Thus, $\text{span}\{a_1, \ldots, a_k\} \subseteq \text{span}\{q_1, \ldots, q_k\}$. However, since $\text{rank}(A) = n$ it follows that $\text{span}\{a_1, \ldots, a_k\}$ has dimension k and so must equal $\text{span}\{q_1, \ldots, q_k\}$ The rest of the theorem follows trivially. □

The matrices $Q_1 = Q(1{:}m, 1{:}n)$ and $Q_2 = Q(1{:}m, n+1{:}m)$ can be easily computed from a factored form representation of Q.

If $A = QR$ is a QR factorization of $A \in \mathbb{R}^{m \times n}$ and $m \geq n$, then we refer to $A = Q(:, 1{:}n)R(1{:}n, 1{:}n)$ as the *thin QR factorization*. The next result addresses the uniqueness issue for the *thin* QR factorization

Theorem 5.2.2 *Suppose $A \in \mathbb{R}^{m \times n}$ has full column rank. The thin QR factorization*

$$A = Q_1 R_1$$

is unique where $Q_1 \in \mathbb{R}^{m \times n}$ has orthonormal columns and R_1 is upper triangular with positive diagonal entries. Moreover, $R_1 = G^T$ where G is the lower triangular Cholesky factor of $A^T A$.

Proof. Since $A^T A = (Q_1 R_1)^T (Q_1 R_1) = R_1^T R_1$ we see that $G = R_1^T$ is the Cholesky factor of $A^T A$. This factor is unique by Theorem 4.2.5. Since $Q_1 = A R_1^{-1}$ it follows that Q_1 is also unique. □

How are Q_1 and R_1 affected by perturbations in A? To answer this question we need to extend the notion of condition to rectangular matrices. Recall from §2.7.3 that the 2-norm condition of a square nonsingular matrix is the ratio of the largest and smallest singular values. For rectangular matrices with full column rank we continue with this definition:

$$A \in \mathbb{R}^{m \times n}, \text{rank}(A) = n \implies \kappa_2(A) \;=\; \frac{\sigma_{max}(A)}{\sigma_{min}(A)}\,.$$

If the columns of A are nearly dependent, then $\kappa_2(A)$ is large. Stewart (1993) has shown that $O(\epsilon)$ relative error in A induces $O(\epsilon\kappa_2(A))$ relative error in R and Q_1.

5.2.7 Classical Gram-Schmidt

We now discuss two alternative methods that can be used to compute the thin QR factorization $A = Q_1 R_1$ directly. If $\text{rank}(A) = n$, then equation (5.2.2) can be solved for q_k:

$$q_k \;=\; \left(a_k \;-\; \sum_{i=1}^{k-1} r_{ik} q_i\right) \Big/ r_{kk}\,.$$

Thus, we can think of q_k as a unit 2-norm vector in the direction of

$$z_k = a_k - \sum_{i=1}^{k-1} r_{ik} q_i$$

where to ensure $z_k \in \text{span}\{q_1, \ldots, q_{k-1}\}^\perp$ we choose

$$r_{ik} = q_i^T a_k \qquad i = 1{:}k-1 .$$

This leads to the *classical Gram-Schmidt* (CGS) algorithm for computing $A = Q_1 R_1$.

```
R(1,1) = || A(:,1) ||_2
Q(:,1) = A(:,1)/R(1,1)
for k = 2:n
    R(1:k-1,k) = Q(1:m,1:k-1)^T A(1:m,k)
    z = A(1:m,k) - Q(1:m,1:k-1)R(1:k-1,k)          (5.2.3)
    R(k,k) = || z ||_2
    Q(1:m,k) = z/R(k,k)
end
```

In the kth step of CGS, the kth columns of both Q and R are generated.

5.2.8 Modified Gram-Schmidt

Unfortunately, the CGS method has very poor numerical properties in that there is typically a severe loss of orthogonality among the computed q_i. Interestingly, a rearrangement of the calculation, known as *modified Gram-Schmidt* (MGS), yields a much sounder computational procedure. In the kth step of MGS, the kth column of Q (denoted by q_k) and the kth row of R (denoted by r_k^T) are determined. To derive the MGS method, define the matrix $A^{(k)} \in \mathbb{R}^{m\times(n-k+1)}$ by

$$A - \sum_{i=1}^{k-1} q_i r_i^T = \sum_{i=k}^{n} q_i r_i^T = [\, 0 \; A^{(k)} \,] . \qquad (5.2.4)$$

It follows that if

$$A^{(k)} = [\; \underset{1}{z} \quad \underset{n-k}{B} \;]$$

then $r_{kk} = \| z \|_2$, $q_k = z/r_{kk}$ and $(r_{k,k+1} \cdots r_{kn}) = q_k^T B$. We then compute the outer product $A^{(k+1)} = B - q_k (r_{k,k+1} \cdots r_{kn})$ and proceed to the next step. This completely describes the kth step of MGS.

Algorithm 5.2.5 (Modified Gram-Schmidt) Given $A \in \mathbb{R}^{m\times n}$ with $\text{rank}(A) = n$, the following algorithm computes the factorization $A = Q_1 R_1$ where $Q_1 \in \mathbb{R}^{m\times n}$ has orthonormal columns and $R_1 \in \mathbb{R}^{n\times n}$ is upper triangular.


```
for k = 1:n
    R(k,k) = || A(1:m,k) ||_2
    Q(1:m,k) = A(1:m,k)/R(k,k)
    for j = k+1:n
        R(k,j) = Q(1:m,k)^T A(1:m,j)
        A(1:m,j) = A(1:m,j) - Q(1:m,k)R(k,j)
    end
end
```

This algorithm requires $2mn^2$ flops. It is not possible to overwrite A with both Q_1 and R_1. Typically, the MGS computation is arranged so that A is overwritten by Q_1 and the matrix R_1 is stored in a separate array.

5.2.9 Work and Accuracy

If one is interested in computing an orthonormal basis for $\text{ran}(A)$, then the Householder approach requires $2mn^2 - 2n^3/3$ flops to get Q in factored form and another $2mn^2 - 2n^3/3$ flops to get the first n columns of Q. (This requires "paying attention" to just the first n columns of Q in (5.1.5).) Therefore, for the problem of finding an orthonormal basis for $\text{ran}(A)$, MGS is about twice as efficient as Householder orthogonalization. However, Björck (1967) has shown that MGS produces a computed $\hat{Q}_1 = [\, \hat{q}_1, \ldots, \hat{q}_n \,]$ that satisfies

$$\hat{Q}_1^T \hat{Q}_1 = I + E_{MGS} \qquad \| E_{MGS} \|_2 \approx \mathbf{u}\kappa_2(A)$$

whereas the corresponding result for the Householder approach is of the form

$$\hat{Q}_1^T \hat{Q}_1 = I + E_H \qquad \| E_H \|_2 \approx \mathbf{u} .$$

Thus, if orthonormality is critical, then MGS should be used to compute orthonormal bases only when the vectors to be orthogonalized are fairly independent.

We also mention that the computed triangular factor $\hat{R}$ produced by MGS satisfies $\| A - \hat{Q}\hat{R} \| \approx \mathbf{u}\| A \|$ and that there exists a Q with perfectly orthonormal columns such that $\| A - Q\hat{R} \| \approx \mathbf{u}\| A \|$. See Higham (1996, p.379).

Example 5.2.1 If modified Gram-Schmidt is applied to

$$A = \begin{bmatrix} 1 & 1 \\ 10^{-3} & 0 \\ 0 & 10^{-3} \end{bmatrix} \qquad \kappa_2(A) \approx 1.4 \cdot 10^3$$

with 6-digit decimal arithmetic, then

$$[\hat{q}_1 \; \hat{q}_2] = \begin{bmatrix} 1.00000 & 0 \\ .001 & -.707107 \\ 0 & .707100 \end{bmatrix} .$$

5.2.10 A Note on Complex QR

Most of the algorithms that we present in this book have complex versions that are fairly straight forward to derive from their real counterparts. (This is *not* to say that everything is easy and obvious at the implementation level.) As an illustration we outline what a complex Householder QR factorization algorithm looks like.

Starting at the level of an individual Householder transformation, suppose $0 \neq x \in \mathbb{C}^n$ and that $x_1 = re^{i\theta}$ where $r, \theta \in \mathbb{R}$. If $v = x \pm e^{i\theta}\| x \|_2 e_1$ and $P = I_n - \beta vv^H$, $\beta = 2/v^H v$, then $Px = \mp e^{i\theta}\| x \|_2 e_1$. (See P5.1.3.) The sign can be determined to maximize $\| v \|_2$ for the sake of stability.

The upper triangularization of $A \in \mathbb{R}^{m \times n}$, $m \geq n$, proceeds as in Algorithm 5.2.1. In step j we zero the subdiagonal portion of $A(j{:}m, j)$:

for $j = 1{:}n$
  $x = A(j{:}m, j)$
  $v = x \pm e^{i\theta}\| x \|_2 e_1$ where $x_1 = re^{i\theta}$.
  $\beta = 2/v^H/v$
  $A(j{:}m, j{:}n) = (I_{m-j+1} - \beta vv^H)A(j{:}m, j{:}n)$
end

The reduction involves $8n^2(m - n/3)$ real flops, four times the number required to execute Algorithm 5.2.1. If $Q = P_1 \cdots P_n$ is the product of the Householder transformations, then Q is unitary and $Q^T A = R \in \mathbb{R}^{m \times n}$ is complex and upper triangular.

Problems

P5.2.1 Adapt the Householder QR algorithm so that it can efficiently handle the case when $A \in \mathbb{R}^{m \times n}$ has lower bandwidth p and upper bandwidth q.

P5.2.2 Adapt the Householder QR algorithm so that it computes the factorization $A = QL$ where L is lower triangular and Q is orthogonal. Assume that A is square. This involves rewriting the Householder vector function $v = \text{house}(x)$ so that $(I - 2vv^T/v^T v)x$ is zero everywhere but its bottom component.

P5.2.3 Adapt the Givens QR factorization algorithm so that the zeros are introduced by diagonal. That is, the entries are zeroed in the order $(m, 1)$, $(m-1, 1)$, $(m, 2)$, $(m-2, 1)$, $(m-1, 2)$, $(m, 3)$, etc.

P5.2.4 Adapt the fast Givens QR factorization algorithm so that it efficiently handles the case when A is n-by-n and tridiagonal. Assume that the subdiagonal, diagonal, and superdiagonal of A are stored in $e(1{:}n-1)$, $a(1{:}n)$, $f(1{:}n-1)$ respectively. Design your algorithm so that these vectors are overwritten by the nonzero portion of T.

P5.2.5 Suppose $L \in \mathbb{R}^{m \times n}$ with $m \geq n$ is lower triangular. Show how Householder matrices $H_1 \ldots H_n$ can be used to determine a lower triangular $L_1 \in \mathbb{R}^{n \times n}$ so that

$$H_n \cdots H_1 L = \begin{bmatrix} L_1 \\ 0 \end{bmatrix}$$

Hint: The second step in the 6-by-3 case involves finding H_2 so that

$$H_2 \begin{bmatrix} \times & 0 & 0 \\ \times & \times & 0 \\ \times & \times & \times \\ \times & \times & 0 \\ \times & \times & 0 \\ \times & \times & 0 \end{bmatrix} = \begin{bmatrix} \times & 0 & 0 \\ \times & \times & 0 \\ \times & \times & \times \\ \times & 0 & 0 \\ \times & 0 & 0 \\ \times & 0 & 0 \end{bmatrix}$$

with the property that rows 1 and 3 are left alone.

P5.2.6 Show that if

$$A = \begin{bmatrix} R & w \\ 0 & v \end{bmatrix} \begin{matrix} k \\ m-k \end{matrix} \qquad b = \begin{bmatrix} c \\ d \end{bmatrix} \begin{matrix} k \\ m-k \end{matrix}$$
$$\quad k \quad n-k$$

and A has full column rank, then $\min \| Ax - b \|_2^2 = \| d \|_2^2 - \left(v^T d/\| v \|_2\right)^2$.

P5.2.7 Suppose $A \in \mathbb{R}^{n \times n}$ and $D = \text{diag}(d_1, \ldots, d_n) \in \mathbb{R}^{n \times n}$. Show how to construct an orthogonal Q such that $Q^T A - DQ^T = R$ is upper triangular. Do not worry about efficiency—this is just an exercise in QR manipulation.

P5.2.8 Show how to compute the QR factorization of the product $A = A_p \cdots A_2 A_1$ without explicitly multiplying the matrices $A_1, \ldots, A_p$ together. Hint: In the $p = 3$ case, write $Q_3^T A = Q_3^T A_3 Q_2 Q_2^T A_2 Q_1 Q_1^T A_1$ and determine orthogonal Q_i so that $Q_i^T(A_i Q_{i-1})$ is upper triangular. ($Q_0 = I$).

P5.2.9 Suppose $A \in \mathbb{R}^{n \times n}$ and let E be the permutation obtained by reversing the order of the rows in I_n. (This is just the exchange matrix of §4.7.) (a) Show that if $R \in \mathbb{R}^{n \times n}$ is upper triangular, then $L = ERE$ is lower triangular. (b) Show how to compute an orthogonal $Q \in \mathbb{R}^{n \times n}$ and a lower triangular $L \in \mathbb{R}^{n \times n}$ so that $A = QL$ assuming the availability of a procedure for computing the QR factorization.

P5.2.10 MGS applied to $A \in \mathbb{R}^{m \times n}$ is numerically equivalent to the first step in Householder QR applied to

$$\tilde{A} = \begin{bmatrix} O_n \\ A \end{bmatrix}$$

where O_n is the n-by-n zero matrix. Verify that this statement is true after the first step of each method is completed.

P5.2.11 Reverse the loop orders in Algorithm 5.2.5 (MGS QR) so that R is computed column-by-column.

P5.2.12 Develop a complex version of the Givens QR factorization. Refer to P5.1.5, where complex Givens rotations are the theme. Is it possible to organize the calculations so that the diagonal elements of R are nonnegative?

Notes and References for Sec. 5.2

The idea of using Householder transformations to solve the LS problem was proposed in

A.S. Householder (1958). "Unitary Triangularization of a Nonsymmetric Matrix," *J. ACM. 5*, 339–42.

The practical details were worked out in

P. Businger and G.H. Golub (1965). "Linear Least Squares Solutions by Householder Transformations," *Numer. Math.* 7, 269–76. See also Wilkinson and Reinsch (1971,111–18).
G.H. Golub (1965). "Numerical Methods for Solving Linear Least Squares Problems," *Numer. Math.* 7, 206–16.

The basic references on QR via Givens rotations include

W. Givens (1958). "Computation of Plane Unitary Rotations Transforming a General Matrix to Triangular Form," *SIAM J. App. Math.* 6, 26–50.

M. Gentleman (1973). "Error Analysis of QR Decompositions by Givens Transformations," *Lin. Alg. and Its Appl.* 10, 189–97.

For a discussion of how the QR factorization can be used to solve numerous problems in statistical computation, see

G.H. Golub (1969). "Matrix Decompositions and Statistical Computation," in *Statistical Computation* , ed. R.C. Milton and J.A. Nelder, Academic Press, New York, pp. 365–97.

The behavior of the Q and R factors when A is perturbed is discussed in

G.W. Stewart (1977). "Perturbation Bounds for the QR Factorization of a Matrix," *SIAM J. Num. Anal.* 14, 509–18.

H. Zha (1993). "A Componentwise Perturbation Analysis of the QR Decomposition," *SIAM J. Matrix Anal. Appl.* 4, 1124–1131.

G.W. Stewart (1993). "On the Perturbation of LU Cholesky, and QR Factorizations," *SIAM J. Matrix Anal. Appl.* 14, 1141–1145.

A. Barrlund (1994). "Perturbation Bounds for the Generalized QR Factorization," *Lin. Alg. and Its Applic.* 207, 251–271.

J.-G. Sun (1995). "On Perturbation Bounds for the QR Factorization," *Lin. Alg. and Its Applic.* 215, 95–112.

The main result is that the changes in Q and R are bounded by the condition of A times the relative change in A. Organizing the computation so that the entries in Q depend continuously on the entries in A is discussed in

T.F. Coleman and D.C. Sorensen (1984). "A Note on the Computation of an Orthonormal Basis for the Null Space of a Matrix," *Mathematical Programming* 29, 234–242.

References for the Gram-Schmidt process include include

J.R. Rice (1966). "Experiments on Gram-Schmidt Orthogonalization," *Math. Comp.* 20, 325–28.

A. Björck (1967). "Solving Linear Least Squares Problems by Gram-Schmidt Orthogonalization," *BIT* 7, 1–21.

N.N. Abdelmalek (1971). "Roundoff Error Analysis for Gram-Schmidt Method and Solution of Linear Least Squares Problems," *BIT* 11, 345–68.

J. Daniel, W.B. Gragg, L.Kaufman, and G.W. Stewart (1976). "Reorthogonalization and Stable Algorithms for Updating the Gram-Schmidt QR Factorization," *Math. Comp.* 30, 772–795.

A. Ruhe (1983). "Numerical Aspects of Gram-Schmidt Orthogonalization of Vectors," *Lin. Alg. and Its Applic.* 52/53, 591–601.

W. Jalby and B. Philippe (1991). "Stability Analysis and Improvement of the Block Gram-Schmidt Algorithm," *SIAM J. Sci. Stat. Comp.* 12, 1058–1073.

Å. Björck and C.C. Paige (1992). "Loss and Recapture of Orthogonality in the Modified Gram-Schmidt Algorithm," *SIAM J. Matrix Anal. Appl.* 13, 176–190.

A. Björck (1994). "Numerics of Gram-Schmidt Orthogonalization," *Lin. Alg. and Its Applic.* 197/198, 297–316.

The QR factorization of a structured matrix is usually structured itself. See

A.W. Bojanczyk, R.P. Brent, and F.R. de Hoog (1986). "QR Factorization of Toeplitz Matrices," *Numer. Math.* 49, 81–94.

S. Qiao(1986). "Hybrid Algorithm for Fast Toeplitz Orthogonalization," *Numer. Math.* 53, 351–366.

C.J. Demeure (1989). "Fast QR Factorization of Vandermonde Matrices," *Lin. Alg. and Its Applic.* 122/123/124, 165–194.

L. Reichel (1991). "Fast QR Decomposition of Vandermonde-Like Matrices and Polynomial Least Squares Approximation," *SIAM J. Matrix Anal. Appl.* 12, 552–564.

D.R. Sweet (1991). "Fast Block Toeplitz Orthogonalization," *Numer. Math.* 58, 613–629.

Various high-performance issues pertaining to the QR factorization are discussed in

B. Mattingly, C. Meyer, and J. Ortega (1989). "Orthogonal Reduction on Vector Computers," *SIAM J. Sci. and Stat. Comp.* 10, 372–381.

P.A. Knight (1995). "Fast Rectangular Matrix Multiplication and the QR Decomposition," *Lin. Alg. and Its Applic.* 221, 69–81.

5.3 The Full Rank LS Problem

Consider the problem of finding a vector $x \in \mathbb{R}^n$ such that $Ax = b$ where the *data matrix* $A \in \mathbb{R}^{m\times n}$ and the *observation vector* $b \in \mathbb{R}^m$ are given and $m \ge n$. When there are more equations than unknowns, we say that the system $Ax = b$ is *overdetermined*. Usually an overdetermined system has no exact solution since b must be an element of ran(A), a proper subspace of $\mathbb{R}^m$.

This suggests that we strive to minimize $\| Ax - b \|_p$ for some suitable choice of p. Different norms render different optimum solutions. For example, if $A = [\,1,\ 1,\ 1\,]^T$ and $b = [\,b_1,\ b_2,\ b_3\,]^T$ with $b_1 \ge b_2 \ge b_3 \ge 0$, then it can be verified that

$$\begin{array}{lllll} p = 1 & \Rightarrow & x_{opt} & = & b_2 \\ p = 2 & \Rightarrow & x_{opt} & = & (b_1 + b_2 + b_3)/3 \\ p = \infty & \Rightarrow & x_{opt} & = & (b_1 + b_3)/2. \end{array}$$

Minimization in the 1-norm and ∞ -norm is complicated by the fact that the function $f(x) = \| Ax - b \|_p$ is not differentiable for these values of p. However, much progress has been made in this area, and there are several good techniques available for 1-norm and ∞-norm minimization. See Coleman and Li (1992), Li (1993), and Zhang (1993).

In contrast to general p-norm minimization, the *least squares* (LS) problem

$$\min_{x\in\mathbb{R}^n} \| Ax - b \|_2 \qquad (5.3.1)$$

is more tractable for two reasons:

- $\phi(x) = \frac{1}{2}\| Ax - b \|_2^2$ is a differentiable function of x and so the minimizers of ϕ satisfy the gradient equation $\nabla\phi(x) = 0$. This turns out to be an easily constructed symmetric linear system which is positive definite if A has full column rank.

- The 2-norm is preserved under orthogonal transformation. This means that we can seek an orthogonal Q such that the equivalent problem of minimizing $\| (Q^TA)x - (Q^Tb) \|_2$ is "easy" to solve.

In this section we pursue these two solution approaches for the case when A has full column rank. Methods based on normal equations and the QR factorization are detailed and compared.

5.3.1 Implications of Full Rank

Suppose $x \in \mathbb{R}^n$, $z \in \mathbb{R}^n$, and $\alpha \in \mathbb{R}$ and consider the equality

$$\| A(x + \alpha z) - b \|_2^2 \; = \; \| Ax - b \|_2^2 \; + 2\alpha z^T A^T(Ax - b) + \alpha^2 \| Az \|_2^2$$

where $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$. If x solves the LS problem (5.3.1) then we must have $A^T(Ax - b) = 0$. Otherwise, if $z = -A^T(Ax - b)$ and we make α small enough, then we obtain the contradictory inequality $\| A(x + \alpha z) - b \|_2 < \| Ax - b \|_2$. We may also conclude that if x *and* $x + \alpha z$ are LS minimizers, then $z \in \text{null}(A)$.

Thus, if A has full column rank, then there is a unique LS solution x_{LS} and it solves the symmetric positive definite linear system

$$A^T A x_{LS} = A^T b \,.$$

These are called the *normal equations.* Since $\nabla \phi(x) = A^T(Ax - b)$ where $\phi(x) = \frac{1}{2} \| Ax - b \|_2^2$, we see that solving the normal equations is tantamount to solving the gradient equation $\nabla \phi = 0$. We call

$$r_{LS} \; = \; b - Ax_{LS}$$

the *minimum residual* and we use the notation

$$\rho_{LS} \; = \; \| Ax_{LS} - b \|_2$$

to denote its size. Note that if ρ_{LS} is small, then we can "predict" b with the columns of A.

So far we have been assuming that $A \in \mathbb{R}^{m \times n}$ has full column rank. This assumption is dropped in §5.5. However, even if $\text{rank}(A) = n$, then we can expect trouble in the above procedures if A is nearly rank deficient.

When assessing the quality of a computed LS solution $\hat{x}_{LS}$, there are two important issues to bear in mind:

- How close is $\hat{x}_{LS}$ to x_{LS}?
- How small is $\hat{r}_{LS} = b - A\hat{x}_{LS}$ compared to $r_{LS} = b - Ax_{LS}$?

The relative importance of these two criteria varies from application to application. In any case it is important to understand how x_{LS} and r_{LS} are affected by perturbations in A and b. Our intuition tells us that if the columns of A are nearly dependent, then these quantities may be quite sensitive.

Example 5.3.1 Suppose

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 10^{-6} \\ 0 & 0 \end{bmatrix}, \; \delta A = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 10^{-8} \end{bmatrix}, \; b = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \; \delta b = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix},$$

and that x_{LS} and $\hat{x}_{LS}$ minimize $\| Ax - b \|_2$ and $\| (A + \delta A)x - (b + \delta b) \|_2$ respectively. Let r_{LS} and $\hat{r}_{LS}$ be the corresponding minimum residuals. Then

$$x_{LS} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \; \hat{x}_{LS} = \begin{bmatrix} 1 \\ .9999 \cdot 10^4 \end{bmatrix}, \; r_{LS} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \; \hat{r}_{LS} = \begin{bmatrix} 0 \\ -.9999 \cdot 10^{-2} \\ .9999 \cdot 10^0 \end{bmatrix}.$$

Since $\kappa_2(A) = 10^6$ we have

$$\frac{\| \hat{x}_{LS} - x_{LS} \|_2}{\| x_{LS} \|_2} \approx .9999 \cdot 10^4 \le \kappa_2(A)^2 \frac{\| \delta A \|_2}{\| A \|_2} = 10^{12} \cdot 10^{-8}$$

and

$$\frac{\| \hat{r}_{LS} - r_{LS} \|_2}{\| b \|_2} \approx .7070 \cdot 10^{-2} \le \kappa_2(A) \frac{\| \delta A \|_2}{\| A \|_2} = 10^6 \cdot 10^{-8}.$$

The example suggests that the sensitivity of x_{LS} depends upon $\kappa_2(A)^2$. At the end of this section we develop a perturbation theory for the LS problem and the $\kappa_2(A)^2$ factor will return.

5.3.2 The Method of Normal Equations

The most widely used method for solving the full rank LS problem is the method of normal equations.

Algorithm 5.3.1 (Normal Equations) Given $A \in \mathbb{R}^{m \times n}$ with the property that $\text{rank}(A) = n$ and $b \in \mathbb{R}^m$, this algorithm computes the solution x_{LS} to the LS problem $\min \| Ax - b \|_2$ where $b \in \mathbb{R}^m$.

Compute the lower triangular portion of $C = A^T A$.
$d = A^T b$
Compute the Cholesky factorization $C = GG^T$.
Solve $Gy = d$ and $G^T x_{LS} = y$.

This algorithm requires $(m + n/3)n^2$ flops. The normal equation approach is convenient because it relies on standard algorithms: Cholesky factorization, matrix-matrix multiplication, and matrix-vector multiplication. The compression of the m-by-n data matrix A into the (typically) much smaller n-by-n cross-product matrix C is attractive.

Let us consider the accuracy of the computed normal equations solution $\hat{x}_{LS}$. For clarity, assume that no roundoff errors occur during the formation of $C = A^TA$ and $d = A^Tb$. (On many computers inner products are accumulated in double precision and so this is not a terribly unfair assumption.) It follows from what we know about the roundoff properties of the Cholesky factorization (cf. §4.2.7) that

$$(A^TA + E)\hat{x}_{LS} \;=\; A^Tb,$$

where $\| E \|_2 \approx \mathbf{u}\| A^T \|_2 \| A \|_2 \approx \mathbf{u}\| A^TA \|_2$ and thus we can expect

$$\frac{\| \hat{x}_{LS} - x_{LS} \|_2}{\| x_{LS} \|_2} \;\approx\; \mathbf{u}\kappa_2(A^TA) \;=\; \mathbf{u}\kappa_2(A)^2. \tag{5.3.2}$$

In other words, the accuracy of the computed normal equations solution depends on the square of the condition. This seems to be consistent with Example 5.3.1 but more refined comments follow in §5.3.9.

Example 5.3.2 It should be noted that the formation of A^TA can result in a severe loss of information.

$$A = \begin{bmatrix} 1 & 1 \\ 10^{-3} & 0 \\ 0 & 10^{-3} \end{bmatrix} \text{ and } b = \begin{bmatrix} 2 \\ 10^{-3} \\ 10^{-3} \end{bmatrix}$$

then $\kappa_2(A) \approx 1.4 \cdot 10^3$, $x_{LS} = [1\ 1]^T$, and $\rho_{LS} = 0$. If the normal equations method is executed with base 10, $t = 6$ arithmetic, then a divide-by-zero occurs during the solution process, since

$$fl(A^TA) = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

is exactly singular. On the other hand, if 7-digit arithmetic is used, then $\hat{x}_{LS} = [\,2.000001\,,\ 0\,]^T$ and $\| \hat{x}_{LS} - x_{LS} \|_2 / \| x_{LS} \|_2 \approx \mathbf{u}\kappa_2(A)^2$.

5.3.3 LS Solution Via QR Factorization

Let $A \in \mathbb{R}^{m\times n}$ with $m \geq n$ and $b \in \mathbb{R}^m$ be given and suppose that an orthogonal matrix $Q \in \mathbb{R}^{m\times m}$ has been computed such that

$$Q^TA = R = \begin{bmatrix} R_1 \\ 0 \end{bmatrix} \begin{matrix} n \\ m-n \end{matrix} \tag{5.3.3}$$

is upper triangular. If

$$Q^Tb = \begin{bmatrix} c \\ d \end{bmatrix} \begin{matrix} n \\ m-n \end{matrix}$$

then

$$\| Ax - b \|_2^2 \;=\; \| Q^TAx - Q^Tb \|_2^2 \;=\; \| R_1x - c \|_2^2 \;+\; \| d \|_2^2$$

for any $x \in \mathbb{R}^n$. Clearly, if $\text{rank}(A) = \text{rank}(R_1) = n$, then x_{LS} is defined by the upper triangular system $R_1x_{LS} = c$. Note that

$$\rho_{LS} = \| d \|_2.$$

We conclude that the full rank LS problem can be readily solved once we have computed the QR factorization of A. Details depend on the exact QR procedure. If Householder matrices are used and Q^T is applied in factored form to b, then we obtain

Algorithm 5.3.2 (Householder LS Solution) If $A \in \mathbb{R}^{m\times n}$ has full column rank and $b \in \mathbb{R}^m$, then the following algorithm computes a vector $x_{LS} \in \mathbb{R}^n$ such that $\| Ax_{LS} - b \|_2$ is minimum.

```
Use Algorithm 5.2.1 to overwrite A with its QR factorization.
for j = 1:n
    v(j) = 1; v(j + 1:m) = A(j + 1:m, j)
    b(j:m) = (I_{m-j+1} - β_j v v^T) b(j:m)
end
Solve R(1:n, 1:n) x_LS = b(1:n) using back substitution.
```

This method for solving the full rank LS problem requires $2n^2(m - n/3)$ flops. The $O(mn)$ flops associated with the updating of b and the $O(n^2)$ flops associated with the back substitution are not significant compared to the work required to factor A.

It can be shown that the computed $\hat{x}_{LS}$ solves

$$\min \| (A + \delta A)x - (b + \delta b) \|_2 \tag{5.3.4}$$

where

$$\| \delta A \|_F \leq (6m - 3n + 41)n\mathbf{u}\| A \|_F + O(\mathbf{u}^2) \tag{5.3.5}$$

and

$$\| \delta b \|_2 \leq (6m - 3n + 40)n\mathbf{u}\| b \|_2 + O(\mathbf{u}^2). \tag{5.3.6}$$

These inequalities are established in Lawson and Hanson (1974, p.90ff) and show that $\hat{x}_{LS}$ satisfies a "nearby" LS problem. (We cannot address the relative error in $\hat{x}_{LS}$ without an LS perturbation theory, to be discussed shortly.) We mention that similar results hold if Givens QR is used.

5.3.4 Breakdown in Near-Rank Deficient Case

Like the method of normal equations, the Householder method for solving the LS problem breaks down in the back substitution phase if $\text{rank}(A) < n$. Numerically, trouble can be expected whenever $\kappa_2(A) = \kappa_2(R) \approx 1/\mathbf{u}$. This is in contrast to the normal equations approach, where completion of the Cholesky factorization becomes problematical once $\kappa_2(A)$ is in the

neighborhood of $1/\sqrt{\mathbf{u}}$. (See Example 5.3.2.) Hence the claim in Lawson and Hanson (1974, 126–127) that for a fixed machine precision, a wider class of LS problems can be solved using Householder orthogonalization.

5.3.5 A Note on the MGS Approach

In principle, MGS computes the thin QR factorization $A = Q_1R_1$. This is enough to solve the full rank LS problem because it transforms the normal equations $(A^TA)x = A^Tb$ to the upper triangular system $R_1x = Q_1^Tb$. But an analysis of this approach when Q_1^Tb is explicitly formed introduces a $\kappa_2(A)^2$ term. This is because the computed factor $\hat{Q}_1$ satisfies $\| \hat{Q}_1^T\hat{Q}_1 - I_n \|_2 \approx \mathbf{u}\kappa_2(A)$ as we mentioned in §5.2.9.

However, if MGS is applied to the augmented matrix

$$A_+ = [\, A\; b \,] = [\, Q_1\; q_{n+1} \,] \begin{bmatrix} R_1 & z \\ 0 & \rho \end{bmatrix},$$

then $z = Q_1^Tb$. Computing Q_1^Tb in this fashion and solving $R_1x_{LS} = z$ produces an LS solution $\hat{x}_{LS}$ that is "just as good" as the Householder QR method. That is to say, a result of the form (5.3.4)-(5.3.6) applies. See Björck and Paige (1992).

It should be noted that the MGS method is slightly more expensive than Householder QR because it always manipulates m-vectors whereas the latter procedure deals with ever shorter vectors.

5.3.6 Fast Givens LS Solver

The LS problem can also be solved using fast Givens transformations. Suppose $M^TM = D$ is diagonal and

$$M^TA = \begin{bmatrix} S_1 \\ 0 \end{bmatrix} \begin{matrix} n \\ m-n \end{matrix}$$

is upper triangular. If

$$M^Tb = \begin{bmatrix} c \\ d \end{bmatrix} \begin{matrix} n \\ m-n \end{matrix}$$

then

$$\| Ax - b \|_2^2 = \| D^{-1/2}M^T(Ax - b) \|_2^2 = \left\| D^{-1/2} \left(\begin{bmatrix} S_1 \\ 0 \end{bmatrix} x - \begin{bmatrix} c \\ d \end{bmatrix} \right) \right\|_2^2$$

for any $x \in \mathbb{R}^n$. Clearly, x_{LS} is obtained by solving the nonsingular upper triangular system $S_1x = c$.

The computed solution $\hat{x}_{LS}$ obtained in this fashion can be shown to solve a nearby LS problem in the sense of (5.3.4)-(5.3.6). This may seem

surprising since large numbers can arise during the calculation. An entry in the scaling matrix D can double in magnitude after a single fast Givens update. However, largeness in D must be exactly compensated for by largeness in M, since $D^{-1/2}M$ is orthogonal at all stages of the computation. It is this phenomenon that enables one to push through a favorable error analysis.

5.3.7 The Sensitivity of the LS Problem

We now develop a perturbation theory that assists in the comparison of the normal equations and QR approaches to the LS problem. The theorem below examines how the LS solution and its residual are affected by changes in A and b. In so doing, the condition of the LS problem is identified.

Two easily established facts are required in the analysis:

$$\begin{aligned} \| A \|_2 \, \| (A^TA)^{-1}A^T \|_2 &= \kappa_2(A) \\ \| A \|_2^2 \, \| (A^TA)^{-1} \|_2 &= \kappa_2(A)^2 \end{aligned} \qquad (5.3.7)$$

These equations can be verified using the SVD.

Theorem 5.3.1 *Suppose x, r, $\hat{x}$, and $\hat{r}$ satisfy*

$$\begin{aligned} \| Ax - b \|_2 &= \min & r &= b - Ax \\ \| (A + \delta A)\hat{x} - (b + \delta b) \|_2 &= \min & \hat{r} &= (b + \delta b) - (A + \delta A)\hat{x} \end{aligned}$$

where A and δA are in $\mathbb{R}^{m \times n}$ with $m \ge n$ and $0 \ne b$ and δb are in $\mathbb{R}^m$. If

$$\epsilon = \max \left\{ \frac{\| \delta A \|_2}{\| A \|_2}, \frac{\| \delta b \|_2}{\| b \|_2} \right\} < \frac{\sigma_n(A)}{\sigma_1(A)}$$

and

$$\sin(\theta) = \frac{\rho_{LS}}{\| b \|_2} \ne 1$$

where $\rho_{LS} = \| Ax_{LS} - b \|_2$, then

$$\frac{\| \hat{x} - x \|_2}{\| x \|_2} \le \epsilon \left\{ \frac{2\kappa_2(A)}{\cos(\theta)} + \tan(\theta)\kappa_2(A)^2 \right\} + O(\epsilon^2) \qquad (5.3.8)$$

$$\frac{\| \hat{r} - r \|_2}{\| b \|_2} \le \epsilon \, (1 + 2\kappa_2(A)) \min(1, m - n) + O(\epsilon^2). \qquad (5.3.9)$$

Proof. Let E and f be defined by $E = \delta A/\epsilon$ and $f = \delta b/\epsilon$. By hypothesis $\| \delta A \|_2 < \sigma_n(A)$ and so by Theorem 2.5.2 we have $\text{rank}(A + tE) = n$ for all $t \in [0, \epsilon]$. It follows that the solution $x(t)$ to

$$(A+tE)^T(A+tE)x(t) \;=\; (A+tE)^T(b+tf) \tag{5.3.10}$$

is continuously differentiable for all $t \in [0, \epsilon]$. Since $x = x(0)$ and $\hat{x} = x(\epsilon)$, we have

$$\hat{x} = x \;+\; \epsilon\dot{x}(0) \;+\; O(\epsilon^2).$$

The assumptions $b \neq 0$ and $\sin(\theta) \neq 1$ ensure that x is nonzero and so

$$\frac{\| \hat{x} - x \|_2}{\| x \|_2} \;=\; \epsilon\,\frac{\| \dot{x}(0) \|_2}{\| x \|_2} \;+\; O(\epsilon^2). \tag{5.3.11}$$

In order to bound $\| \dot{x}(0) \|_2$, we differentiate (5.3.10) and set $t = 0$ in the result. This gives

$$E^TAx + A^TEx + A^TA\dot{x}(0) \;=\; A^Tf + E^Tb$$

i.e.,

$$\dot{x}(0) \;=\; (A^TA)^{-1}A^T(f - Ex) + (A^TA)^{-1}E^Tr. \tag{5.3.12}$$

By substituting this result into (5.3.11), taking norms, and using the easily verified inequalities $\| f \|_2 \le \| b \|_2$ and $\| E \|_2 \le \| A \|_2$ we obtain

$$\begin{aligned}\frac{\| \hat{x} - x \|_2}{\| x \|_2} \;&\le\; \epsilon\left\{\| A \|_2 \| (A^TA)^{-1}A^T \|_2 \left(\frac{\| b \|_2}{\| A \|_2 \| x \|_2} \;+\; 1\right)\right. \\ &\qquad \left. +\; \frac{\rho_{LS}}{\| A \|_2 \| x \|_2} \| A \|_2^2 \, \| (A^TA)^{-1} \|_2 \right\} \;+\; O(\epsilon^2).\end{aligned}$$

Since $A^T(Ax - b) = 0$, Ax is orthogonal to $Ax - b$ and so

$$\| b - Ax \|_2^2 \;+\; \| Ax \|_2^2 \;=\; \| b \|_2^2 .$$

Thus,

$$\| A \|_2^2 \, \| x \|_2^2 \;\ge\; \| b \|_2^2 \;-\; \rho_{LS}^2$$

and so by using (5.3.7)

$$\frac{\| \hat{x} - x \|_2}{\| x \|_2} \;\le\; \epsilon\left\{\kappa_2(A)\left(\frac{1}{\cos(\theta)} \;+\; 1\right) \;+\; \kappa_2(A)^2\frac{\sin(\theta)}{\cos(\theta)}\right\} \;+\; O(\epsilon^2)$$

thereby establishing (5.3.8).

To prove (5.3.9), we define the differentiable vector function $r(t)$ by

$$r(t) = (b + tf) - (A + tE)x(t)$$

and observe that $r = r(0)$ and $\hat{r} = r(\epsilon)$. Using (5.3.12) it can be shown that

$$\dot{r}(0) \;=\; \left(I - A(A^TA)^{-1}A^T\right)(f - Ex) - A(A^TA)^{-1}E^Tr\,.$$

Since $\| \hat{r} - r \|_2 = \epsilon \| \dot{r}(0) \|_2 + O(\epsilon^2)$ we have

$$\begin{aligned}\frac{\| \hat{r} - r \|_2}{\| b \|_2} \;&=\; \epsilon\frac{\| \dot{r}(0) \|_2}{\| b \|_2} \;+\; O(\epsilon^2) \\ &\le\; \epsilon\left\{\| I - A(A^TA)^{-1}A^T \|_2 \left(1 \;+\; \frac{\| A \|_2 \| x \|_2}{\| b \|_2}\right)\right. \\ &\qquad \left. +\; \| A(A^TA)^{-1} \|_2 \| A \|_2 \frac{\rho_{LS}}{\| b \|_2}\right\} \;+\; O(\epsilon^2)\,.\end{aligned}$$

Inequality (5.3.9) now follows because

$$\| A \|_2 \| x \|_2 = \| A \|_2 \| A^+b \|_2 \le \kappa_2(A) \| b \|_2\,,$$

$$\rho_{LS} = \| (I - A(A^TA)^{-1}A^T)b \|_2 \;\le\; \| I - A(A^TA)^{-1}A^T \|_2 \| b \|_2\,,$$

and

$$\| (I - A(A^TA)^{-1}A^T \|_2 = \min(m - n, 1). \;\square$$

An interesting feature of the upper bound in (5.3.8) is the factor

$$\tan(\theta)\kappa_2(A)^2 \;=\; \frac{\rho_{LS}}{\sqrt{\| b \|_2^2 \;-\; \rho_{LS}^2}}\,\kappa_2(A)^2\,.$$

Thus, in nonzero residual problems it is the square of the condition that measures the sensitivity of x_{LS}. In contrast, residual sensitivity depends just linearly on $\kappa_2(A)$. These dependencies are confirmed by Example 5.3.1.

5.3.8 Normal Equations Versus QR

It is instructive to compare the normal equation and QR approaches to the LS problem. Recall the following main points from our discussion:

- The sensitivity of the LS solution is roughly proportional to the quantity $\kappa_2(A) + \rho_{LS}\kappa_2(A)^2$.
- The method of normal equations produces an $\hat{x}_{LS}$ whose relative error depends on the square of the condition.
- The QR approach (Householder, Givens, careful MGS) solves a nearby LS problem and therefore produces a solution that has a relative error approximately given by $\mathbf{u}(\kappa_2(A) + \rho_{LS}\kappa_2(A)^2)$.

Thus, we may conclude that if ρ_{LS} is small and $\kappa_2(A)$ is large, then the method of normal equations does not solve a nearby problem and will usually render an LS solution that is less accurate than a stable QR approach. Conversely, the two methods produce comparably inaccurate results when applied to large residual, ill-conditioned problems.

Finally, we mention two other factors that figure in the debate about QR versus normal equations:

- The normal equations approach involves about half of the arithmetic when $m \gg n$ and does not require as much storage.
- QR approaches are applicable to a wider class of matrices because the Cholesky process applied to A^TA breaks down "before" the back substitution process on $Q^TA = R$.

At the very minimum, this discussion should convince you how difficult it can be to choose the "right" algorithm!

Problems

P5.3.1 Assume $A^TAx = A^Tb$, $(A^TA + F)\hat{x} = A^Tb$, and $2\| F \|_2 \le \sigma_n(A)^2$. Show that if $r = b - Ax$ and $\hat{r} = b - A\hat{x}$, then $\hat{r} - r = A(A^TA + F)^{-1}Fx$ and

$$\| \hat{r} - r \|_2 \le 2\kappa_2(A)\frac{\| F \|_2}{\| A \|_2}\| x \|_2.$$

P5.3.2 Assume that $A^TAx = A^Tb$ and that $A^TA\hat{x} = A^Tb + f$ where $\| f \|_2 \le cu\| A^T \|_2\| b \|_2$ and A has full column rank. Show that

$$\frac{\| x - \hat{x} \|_2}{\| x \|_2} \le cu\kappa_2(A)^2\frac{\| A^T \|_2\| b \|_2}{\| A^Tb \|}.$$

P5.3.3 Let $A \in \mathbb{R}^{m \times n}$ with $m > n$ and $y \in \mathbb{R}^m$ and define $\bar{A} = [A\ y] \in \mathbb{R}^{m \times (n+1)}$. Show that $\sigma_1(\bar{A}) \ge \sigma_1(A)$ and $\sigma_{n+1}(\bar{A}) \le \sigma_n(A)$. Thus, the condition grows if a column is added to a matrix.

P5.3.4 Let $A \in \mathbb{R}^{m \times n}$ $(m \ge n)$, $w \in \mathbb{R}^n$, and define

$$B = \begin{bmatrix} A \\ w^T \end{bmatrix}.$$

Show that $\sigma_n(B) \ge \sigma_n(A)$ and $\sigma_1(B) \le \sqrt{\| A \|_2^2 + \| w \|_2^2}$. Thus, the condition of a matrix may increase or decrease if a row is added.

P5.3.5 (Cline 1973) Suppose that $A \in \mathbb{R}^{m \times n}$ has rank n and that Gaussian elimination with partial pivoting is used to compute the factorization $PA = LU$, where $L \in \mathbb{R}^{m \times n}$ is unit lower triangular, $U \in \mathbb{R}^{n \times n}$ is upper triangular, and $P \in \mathbb{R}^{m \times m}$ is a permutation. Explain how the decomposition in P5.2.5 can be used to find a vector $z \in \mathbb{R}^n$ such that $\| Lz - Pb \|_2$ is minimized. Show that if $Ux = z$, then $\| Ax - b \|_2$ is minimum. Show that this method of solving the LS problem is more efficient than Householder QR from the flop point of view whenever $m \le 5n/3$.

P5.3.6 The matrix $C = (A^TA)^{-1}$, where $\text{rank}(A) = n$, arises in many statistical applications and is known as the *variance-covariance matrix*. Assume that the factorization

$A = QR$ is available. (a) Show $C = (R^TR)^{-1}$. (b) Give an algorithm for computing the diagonal of C that requires $n^3/3$ flops. (c) Show that

$$R = \begin{bmatrix} \alpha & v^T \\ 0 & S \end{bmatrix} \quad \Rightarrow \quad C = (R^TR)^{-1} = \begin{bmatrix} (1 + v^TC_1v)/\alpha^2 & -v^TC_1/\alpha \\ -C_1v/\alpha & C_1 \end{bmatrix}$$

where $C_1 = (S^TS)^{-1}$. (d) Using (c), give an algorithm that overwrites the upper triangular portion of R with the upper triangular portion of C. Your algorithm should require $2n^3/3$ flops.

P5.3.7 Suppose $A \in \mathbb{R}^{n \times n}$ is symmetric and that $r = b - Ax$ where $r, b, x \in \mathbb{R}^n$ and x is nonzero. Show how to compute a symmetric $E \in \mathbb{R}^{n \times n}$ with minimal Frobenius norm so that $(A + E)x = b$. Hint. Use the QR factorization of $[\,x\,,\,r\,]$ and note that $Ex = r \Rightarrow (Q^TEQ)(Q^Tx) = Q^Tr$.

P5.3.8 Show how to compute the nearest circulant matrix to a given Toeplitz matrix. Measure distance with the Frobenius norm.

Notes and References for Sec. 5.3

Our restriction to least squares approximation is not a vote against minimization in other norms. There are occasions when it is advisable to minimize $\| Ax - b \|_p$ for $p = 1$ and ∞. Some algorithms for doing this are described in

A.K. Cline (1976a). "A Descent Method for the Uniform Solution to Overdetermined Systems of Equations," *SIAM J. Num. Anal. 13*, 293–309.

R.H. Bartels, A.R. Conn, and C. Charalambous (1978). "On Cline's Direct Method for Solving Overdetermined Linear Systems in the L_∞ Sense," *SIAM J. Num. Anal. 15*, 255–70.

T.F. Coleman and Y. Li (1992). "A Globally and Quadratically Convergent Affine Scaling Method for Linear L_1 Problems," *Mathematical Programming, 56, Series A*, 189–222.

Y. Li (1993). "A Globally Convergent Method for L_p Problems," *SIAM J. Optimization 3*, 609–629.

Y. Zhang (1993). "A Primal-Dual Interior Point Approach for Computing the L_1 and L_∞ Solutions of Overdetermined Linear Systems," *J. Optimization Theory and Applications 77*, 323–341.

The use of Gauss transformations to solve the LS problem has attracted some attention because they are cheaper to use than Householder or Givens matrices. See

G. Peters and J.H. Wilkinson (1970). "The Least Squares Problem and Pseudo-Inverses," *Comp. J. 13*, 309–16.

A.K. Cline (1973). "An Elimination Method for the Solution of Linear Least Squares Problems," *SIAM J. Num. Anal. 10*, 283–89.

R.J. Plemmons (1974). "Linear Least Squares by Elimination and MGS," *J. Assoc. Comp. Mach. 21*, 581–85.

Important analyses of the LS problem and various solution approaches include

G.H. Golub and J.H. Wilkinson (1966). "Note on the Iterative Refinement of Least Squares Solution," *Numer. Math. 9*, 139–48.

A. van der Sluis (1975). "Stability of the Solutions of Linear Least Squares Problem," *Numer. Math. 23*, 241–54.

Y. Saad (1986). "On the Condition Number of Some Gram Matrices Arising from Least Squares Approximation in the Complex Plane," *Numer. Math. 48*, 337–348.

Å. Björck (1987). "Stability Analysis of the Method of Seminormal Equations," *Lin. Alg. and Its Applic. 88/89*, 31–48.

J. Gluchowska and A. Smoktunowicz (1990). "Solving the Linear Least Squares Problem with Very High Relative Accuracy," *Computing 45*, 345–354.
Å. Björck (1991). "Component-wise Perturbation Analysis and Error Bounds for Linear Least Squares Solutions," *BIT 31*, 238–244.
Å. Björck and C.C. Paige (1992). "Loss and Recapture of Orthogonality in the Modified Gram-Schmidt Algorithm," *SIAM J. Matrix Anal. Appl. 13*, 176–190.
B. Waldén, R. Karlson, J. Sun (1995). "Optimal Backward Perturbation Bounds for the Linear Least Squares Problem," *Numerical Lin. Alg. with Applic. 2*, 271–286.

The "seminormal" equations are given by $R^TRx = A^Tb$ where $A = QR$. In the above paper it is shown that by solving the seminormal equations an acceptable LS solution is obtained if one step of fixed precision iterative improvement is performed.

An Algol implementation of the MGS method for solving the LS problem appears in

F.L. Bauer (1965). "Elimination with Weighted Row Combinations for Solving Linear Equations and Least Squares Problems," *Numer. Math.* 7, 338–52. See also Wilkinson and Reinsch (1971, 119–33).

Least squares problems often have special structure which, of course, should be exploited.

M.G. Cox (1981). "The Least Squares Solution of Overdetermined Linear Equations having Band or Augmented Band Structure," *IMA J. Num. Anal. 1*, 3–22.
G. Cybenko (1984). "The Numerical Stability of the Lattice Algorithm for Least Squares Linear Prediction Problems," *BIT 24*, 441–455.
P.C. Hansen and H. Gesmar (1993). "Fast Orthogonal Decomposition of Rank-Deficient Toeplitz Matrices," *Numerical Algorithms 4*, 151–166.

The use of Householder matrices to solve sparse LS problems requires careful attention to avoid excessive fill-in.

J.K. Reid (1967). "A Note on the Least Squares Solution of a Band System of Linear Equations by Householder Reductions," *Comp. J. 10*, 188–89.
I.S. Duff and J.K. Reid (1976). "A Comparison of Some Methods for the Solution of Sparse Over-Determined Systems of Linear Equations," *J. Inst. Math. Applic. 17*, 267–80.
P.E. Gill and W. Murray (1976). "The Orthogonal Factorization of a Large Sparse Matrix," in *Sparse Matrix Computations*, ed. J.R. Bunch and D.J. Rose, Academic Press, New York, pp. 177-200.
L. Kaufman (1979). "Application of Dense Householder Transformations to a Sparse Matrix," *ACM Trans. Math. Soft. 5*, 442–51.

Although the computation of the QR factorization is more efficient with Householder reflections, there are some settings where the Givens approach is advantageous. For example, if A is sparse, then the careful application of Givens rotations can minimize fill-in.

I.S. Duff (1974). "Pivot Selection and Row Ordering in Givens Reduction on Sparse Matrices," *Computing 13*, 239–48.
J.A. George and M.T. Heath (1980). "Solution of Sparse Linear Least Squares Problems Using Givens Rotations," *Lin. Alg. and Its Applic. 34*, 69–83.

5.4 Other Orthogonal Factorizations

If A is rank deficient, then the QR factorization need not give a basis for $\text{ran}(A)$. This problem can be corrected by computing the QR factorization of a column-permuted version of A, i.e., $A\Pi = QR$ where Π is a permutation.

The "data" in A can be compressed further if we permit right multiplication by a general orthogonal matrix Z:

$$Q^TAZ = T.$$

There are interesting choices for Q and Z and these, together with the column pivoted QR factorization, are discussed in this section.

5.4.1 Rank Deficiency: QR with Column Pivoting

If $A \in \mathbb{R}^{m\times n}$ and $\text{rank}(A) < n$, then the QR factorization does not necessarily produce an orthonormal basis for $\text{ran}(A)$. For example, if A has three columns and

$$A = [\,a_1,\,a_2,\,a_3\,] = [\,q_1,\,q_2,\,q_3\,]\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

is its QR factorization, then $\text{rank}(A) = 2$ but $\text{ran}(A)$ does not equal any of the subspaces $\text{span}\{q_1, q_2\}$, $\text{span}\{q_1, q_3\}$, or $\text{span}\{q_2, q_3\}$.

Fortunately, the Householder QR factorization procedure (Algorithm 5.2.1) can be modified in a simple way to produce an orthonormal basis for $\text{ran}(A)$. The modified algorithm computes the factorization

$$Q^TA\Pi = \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix}\begin{matrix} r \\ m-r \end{matrix} \qquad \begin{matrix} r & n-r \end{matrix} \tag{5.4.1}$$

where $r = \text{rank}(A)$, Q is orthogonal, R_{11} is upper triangular and nonsingular, and Π is a permutation. If we have the column partitionings $A\Pi = [\,a_{c_1}, \ldots, a_{c_n}\,]$ and $Q = [\,q_1, \ldots, q_m\,]$, then for $k = 1{:}n$ we have

$$a_{c_k} = \sum_{i=1}^{\min\{r,k\}} r_{ik}q_i \in \text{span}\{q_1, \ldots, q_r\}$$

implying

$$\text{ran}(A) = \text{span}\{q_1, \ldots, q_r\}.$$

The matrices Q and Π are products of Householder matrices and interchange matrices respectively. Assume for some k that we have computed

Householder matrices $H_1, \ldots, H_{k-1}$ and permutations $\Pi_1, \ldots, \Pi_{k-1}$ such that

$$(H_{k-1} \cdots H_1) A (\Pi_1 \cdots \Pi_{k-1}) = \qquad (5.4.2)$$

$$R^{(k-1)} = \begin{bmatrix} R_{11}^{(k-1)} & R_{12}^{(k-1)} \\ 0 & R_{22}^{(k-1)} \end{bmatrix} \begin{matrix} k-1 \\ m-k+1 \end{matrix}$$
$$\qquad\qquad k-1 \quad n-k+1$$

where $R_{11}^{(k-1)}$ is a nonsingular and upper triangular matrix. Now suppose that

$$R_{22}^{(k-1)} = \left[z_k^{(k-1)}, \ldots, z_n^{(k-1)} \right]$$

is a column partitioning and let $p \geq k$ be the smallest index such that

$$\| z_p^{(k-1)} \|_2 = \max \left\{ \| z_k^{(k-1)} \|_2, \cdots, \| z_n^{(k-1)} \|_2 \right\} . \qquad (5.4.3)$$

Note that if $k-1 = \text{rank}(A)$, then this maximum is zero and we are finished. Otherwise, let Π_k be the n-by-n identity with columns p and k interchanged and determine a Householder matrix H_k such that if $R^{(k)} = H_k R^{(k-1)} \Pi_k$, then $R^{(k)}(k+1{:}m, k) = 0$. In other words, Π_k moves the largest column in $R_{22}^{(k-1)}$ to the lead position and $\tilde{H}_k$ zeroes all of its subdiagonal components.

The column norms do not have to be recomputed at each stage if we exploit the property

$$Q^T z = \begin{bmatrix} \alpha \\ w \end{bmatrix} \begin{matrix} 1 \\ s-1 \end{matrix} \qquad \Longrightarrow \qquad \| w \|_2^2 = \| z \|_2^2 - \alpha^2,$$

which holds for any orthogonal matrix $Q \in \mathbb{R}^{s \times s}$. This reduces the overhead associated with column pivoting from $O(mn^2)$ flops to $O(mn)$ flops because we can get the new column norms by updating the old column norms, e.g.,

$$\| z^{(j)} \|_2^2 = \| z^{(j-1)} \|_2^2 - r_{kj}^2 .$$

Combining all of the above we obtain the following algorithm established by Businger and Golub (1965):

Algorithm 5.4.1 (Householder QR With Column Pivoting) Given $A \in \mathbb{R}^{m \times n}$ with $m \geq n$, the following algorithm computes $r = \text{rank}(A)$ and the factorization (5.4.1) with $Q = H_1 \cdots H_r$ and $\Pi = \Pi_1 \cdots \Pi_r$. The upper triangular part of A is overwritten by the upper triangular part of R and components $j+1{:}m$ of the jth Householder vector are stored in $A(j+1{:}m, j)$. The permutation Π is encoded in an integer vector piv. In particular, Π_j is the identity with rows j and $piv(j)$ interchanged.


```
for j = 1:n
    c(j) = A(1:m,j)^T A(1:m,j)
end
r = 0; τ = max{c(1),...,c(n)}
Find smallest k with 1 ≤ k ≤ n so c(k) = τ
while τ > 0
    r = r + 1
    piv(r) = k; A(1:m,r) ↔ A(1:m,k); c(r) ↔ c(k)
    [v,β] = house(A(r:m,r))
    A(r:m,r:n) = (I_{m-r+1} - βvv^T)A(r:m,r:n)
    A(r+1:m,r) = v(2:m-r+1)
    for i = r+1:n
        c(i) = c(i) - A(r,i)^2
    end
    if r < n
        τ = max{c(r+1),...,c(n)}
        Find smallest k with r+1 ≤ k ≤ n so c(k) = τ.
    else
        τ = 0
    end
end
```

This algorithm requires $4mnr - 2r^2(m+n) + 4r^3/3$ flops where $r = \text{rank}(A)$. As with the nonpivoting procedure, Algorithm 5.2.1, the orthogonal matrix Q is stored in factored form in the subdiagonal portion of A.

Example 5.4.1 If Algorithm 5.4.1 is applied to

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 5 & 6 \\ 1 & 8 & 9 \\ 1 & 11 & 12 \end{bmatrix},$$

then $\Pi = [e_3 \; e_2 \; e_1]$ and to three significant digits we obtain

$$A\Pi = QR = \begin{bmatrix} -.182 & -.816 & .514 & .191 \\ -.365 & .408 & -.827 & .129 \\ .548 & .000 & .113 & -.829 \\ -.730 & .408 & .200 & .510 \end{bmatrix} \begin{bmatrix} -16.4 & -14.600 & -1.820 \\ 0.0 & .816 & -.816 \\ 0.0 & .000 & 0.000 \end{bmatrix} .$$

5.4.2 Complete Orthogonal Decompositions

The matrix R produced by Algorithm 5.4.1 can be further reduced if it is post-multiplied by an appropriate sequence of Householder matrices. In particular, we can use Algorithm 5.2.1 to compute

$$Z_r \cdots Z_1 \begin{bmatrix} R_{11}^T \\ R_{12}^T \end{bmatrix} = \begin{bmatrix} T_{11}^T \\ 0 \end{bmatrix} \begin{matrix} r \\ n-r \end{matrix} \qquad (5.4.4)$$

where the Z_i are Householder transformations and T_{11}^T is upper triangular. It then follows that

$$Q^TAZ = T = \begin{bmatrix} T_{11} & 0 \\ 0 & 0 \end{bmatrix} \begin{matrix} r \\ m-r \end{matrix} \qquad (5.4.5)$$
$$\qquad\quad r \quad n-r$$

where $Z = \Pi Z_1 \cdots Z_r$. We refer to any decomposition of this form as a *complete orthogonal decomposition*. Note that $\text{null}(A) = \text{ran}(Z(1{:}n, r+1{:}n))$. See P5.2.5 for details about the exploitation of structure in (5.4.4).

5.4.3 Bidiagonalization

Suppose $A \in \mathbb{R}^{m\times n}$ and $m \geq n$. We next show how to compute orthogonal U_B (m-by-m) and V_B (n-by-n) such that

$$U_B^TAV_B = \left[\begin{array}{cccccc} d_1 & f_1 & 0 & \cdots & 0 \\ 0 & d_2 & f_2 & & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & & d_{n-1} & f_{n-1} \\ 0 & \cdots & & 0 & d_n \\ \hline & & 0 & & \end{array}\right]. \qquad (5.4.6)$$

$U_B = U_1 \cdots U_n$ and $V_B = V_1 \cdots V_{n-2}$ can each be determined as a product of Householder matrices:

$$\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix} \xrightarrow{U_1} \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix} \xrightarrow{V_1}$$

$$\begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix} \xrightarrow{U_2} \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} \xrightarrow{V_2}$$

$$\begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} \xrightarrow{U_3} \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \\ 0 & 0 & 0 & \times \end{bmatrix} \xrightarrow{U_4} \begin{bmatrix} \times & \times & 0 & 0 \\ 0 & \times & \times & 0 \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

In general, U_k introduces zeros into the kth column, while V_k zeros the appropriate entries in row k. Overall we have:

Algorithm 5.4.2 (Householder Bidiagonalization) Given $A \in \mathbb{R}^{m\times n}$ with $m \geq n$, the following algorithm overwrites A with $U_B^TAV_B = B$ where B is upper bidiagonal and $U_B = U_1 \cdots U_n$ and $V_B = V_1 \cdots V_{n-2}$. The essential part of U_j's Householder vector is stored in $A(j+1{:}m, j)$ and the essential part of V_j's Householder vector is stored in $A(j, j+2{:}n)$.

```
for j = 1:n
    [v, β] = house(A(j:m, j))
    A(j:m, j:n) = (I_{m-j+1} - βvv^T)A(j:m, j:n)
    A(j+1:m, j) = v(2:m - j + 1)
    if j ≤ n - 2
        [v, β] = house(A(j, j+1:n)^T)
        A(j:m, j+1:n) = A(j:m, j+1:n)(I_{n-j} - βvv^T)
        A(j, j+2:n) = v(2:n - j)^T
    end
end
```

This algorithm requires $4mn^2 - 4n^3/3$ flops. Such a technique is used in Golub and Kahan (1965), where bidiagonalization is first described. If the matrices U_B and V_B are explicitly desired, then they can be accumulated in $4m^2n - 4n^3/3$ and $4n^3/3$ flops, respectively. The bidiagonalization of A is related to the tridiagonalization of A^TA. See §8.2.1.

Example 5.4.2 If Algorithm 5.4.2 is applied to

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix},$$

then to three significant digits we obtain

$$\hat{B} = \begin{bmatrix} 12.8 & 21.8 & 0 \\ 0 & 2.24 & -.613 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \hat{V}_B = \begin{bmatrix} 1.00 & 0.00 & 0.00 \\ 0.00 & -.667 & -.745 \\ 0.00 & -.745 & .667 \end{bmatrix}$$

$$\hat{U}_B = \begin{bmatrix} -.0776 & -.833 & .392 & -.383 \\ -.3110 & -.451 & -.238 & .802 \\ -.5430 & -.069 & .701 & -.457 \\ -.7760 & .312 & .547 & .037 \end{bmatrix}.$$

5.4.4 R-Bidiagonalization

A faster method of bidiagonalizing when $m \gg n$ results if we upper triangularize A first before applying Algorithm 5.4.2. In particular, suppose we

compute an orthogonal $Q \in \mathbb{R}^{m\times m}$ such that

$$Q^T A = \begin{bmatrix} R_1 \\ 0 \end{bmatrix}$$

is upper triangular. We then bidiagonalize the square matrix R_1,

$$U_R^T R_1 V_B = B_1 .$$

Here U_R and V_B are n-by-n orthogonal and B_1 is n-by-n upper bidiagonal. If $U_B = Q\,\text{diag}\,(U_R, I_{m-n})$ then

$$U^T AV = \begin{bmatrix} B_1 \\ 0 \end{bmatrix} \equiv B$$

is a bidiagonalization of A.

The idea of computing the bidiagonalization in this manner is mentioned in Lawson and Hanson (1974, p.119) and more fully analyzed in Chan (1982a). We refer to this method as R-bidiagonalization. By comparing its flop count ($2mn^2+2n^3$) with that for Algorithm 5.4.2 ($4mn^2-4n^3/3$) we see that it involves fewer computations (approximately) whenever $m \geq 5n/3$.

5.4.5 The SVD and its Computation

Once the bidiagonalization of A has been achieved, the next step in the Golub-Reinsch SVD algorithm is to zero the superdiagonal elements in B. This is an iterative process and is accomplished by an algorithm due to Golub and Kahan (1965). Unfortunately, we must defer our discussion of this iteration until §8.6 as it requires an understanding of the symmetric eigenvalue problem. Suffice it to say here that it computes orthogonal matrices U_Σ and V_Σ such that

$$U_\Sigma^T B V_\Sigma \;=\; \Sigma \;=\; \text{diag}(\sigma_1,\ldots,\sigma_n) \;\in\; \mathbb{R}^{m\times n} .$$

By defining $U = U_B U_\Sigma$ and $V = V_B V_\Sigma$ we see that $U^T AV = \Sigma$ is the SVD of A. The flop counts associated with this portion of the algorithm depend upon "how much" of the SVD is required. For example, when solving the LS problem, U^T need never be explicitly formed but merely applied to b as it is developed. In other applications, only the matrix $U_1 = U(:,1:n)$ is required. Altogether there are six possibilities and the total amount of work required by the SVD algorithm in each case is summarized in the table below. Because of the two possible bidiagonalization schemes, there are two columns of flop counts. If the bidiagonalization is achieved via Algorithm 5.4.2, the Golub-Reinsch (1970) SVD algorithm results, while if R-bidiagonalization is invoked we obtain the R-SVD algorithm detailed in Chan (1982a). By comparing the entries in this table (which are meant only as approximate estimates of work), we conclude that the R-SVD approach is more efficient unless $m \approx n$.

Required	Golub-Reinsch SVD	R-SVD
Σ	$4mn^2 - 4n^3/3$	$2mn^2 + 2n^3$
Σ, V	$4mn^2 + 8n^3$	$2mn^2 + 11n^3$
Σ, U	$4m^2n - 8mn^2$	$4m^2n + 13n^3$
Σ, U_1	$14mn^2 - 2n^3$	$6mn^2 + 11n^3$
Σ, U, V	$4m^2n + 8mn^2 + 9n^3$	$4m^2n + 22n^3$
Σ, U_1, V	$14mn^2 + 8n^3$	$6mn^2 + 20n^3$

Problems

P5.4.1 Suppose $A \in \mathbb{R}^{m\times n}$ with $m < n$. Give an algorithm for computing the factorization

$$U^T AV = [\,B\;O\,]$$

where B is an m-by-m upper bidiagonal matrix. (Hint: Obtain the form

$$\begin{bmatrix} \times & \times & 0 & 0 & 0 & 0 \\ 0 & \times & \times & 0 & 0 & 0 \\ 0 & 0 & \times & \times & 0 & 0 \\ 0 & 0 & 0 & \times & \times & 0 \end{bmatrix} .$$

using Householder matrices and then "chase" the $(m, m+1)$ entry up the $(m+1)$st column by applying Givens rotations from the right.)

P5.4.2 Show how to efficiently bidiagonalize an n-by-n upper triangular matrix using Givens rotations.

P5.4.3 Show how to upper bidiagonalize a tridiagonal matrix $T \in \mathbb{R}^{n\times n}$ using Givens rotations.

P5.4.4 Let $A \in \mathbb{R}^{m\times n}$ and assume that $0 \neq v$ satisfies $\| Av \|_2 = \sigma_n(A)\| v \|_2$ Let Π be a permutation such that if $\Pi^T v = w$, then $|w_n| = \| w \|_\infty$. Show that if $A\Pi = QR$ is the QR factorization of $A\Pi$, then $|r_{nn}| \leq \sqrt{n}\sigma_n(A)$. Thus, there always exists a permutation Π such that the QR factorization of $A\Pi$ "displays" near rank deficiency.

P5.4.5 Let $x, y \in \mathbb{R}^m$ and $Q \in \mathbb{R}^{m\times m}$ be given with Q orthogonal. Show that if

$$Q^T x \;=\; \begin{bmatrix} \alpha \\ u \end{bmatrix} \begin{matrix} 1 \\ m-1 \end{matrix} \qquad Q^T y \;=\; \begin{bmatrix} \beta \\ v \end{bmatrix} \begin{matrix} 1 \\ m-1 \end{matrix}$$

then $u^T v = x^T y - \alpha\beta$.

P5.4.6 Let $A = [\,a_1,\ldots,a_n\,] \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^m$ be given. For any subset of A's columns $\{a_{c_1},\ldots,a_{c_k}\}$ define

$$\text{res}\,[\,a_{c_1},\ldots,a_{c_k}\,] \;=\; \min_{x\in\mathbb{R}^k} \;\|\,[\,a_{c_1},\ldots,a_{c_k}\,]x - b\,\|_2$$

Describe an alternative pivot selection procedure for Algorithm 5.4.1 such that if $QR = A\Pi = [\,a_{c_1},\ldots,a_{c_n}\,]$ in the final factorization, then for $k = 1:n$:

$$\text{res}\,[\,a_{c_1},\ldots,a_{c_k}\,] = \min_{i\geq k}\; \text{res}[a_{c_1},\ldots,a_{c_{k-1}},a_{c_i}]$$

Notes and References for Sec. 5.4

Aspects of the complete orthogonal decomposition are discussed in

R.J. Hanson and C.L. Lawson (1969). "Extensions and Applications of the Householder Algorithm for Solving Linear Least Square Problems," *Math. Comp.* *23*, 787–812.

P.A. Wedin (1973). "On the Almost Rank-Deficient Case of the Least Squares Problem," *BIT 13*, 344–54.

G.H. Golub and V. Pereyra (1976). "Differentiation of Pseudo-Inverses, Separable Nonlinear Least Squares Problems and Other Tales," in *Generalized Inverses and Applications* , ed. M.Z. Nashed, Academic Press, New York, pp. 303–24.

The computation of the SVD is detailed in §8.6. But here are some of the standard references concerned with its calculation:

G.H. Golub and W. Kahan (1965). "Calculating the Singular Values and Pseudo-Inverse of a Matrix," *SIAM J. Num. Anal. 2*, 205–24.

P.A. Businger and G.H. Golub (1969). "Algorithm 358: Singular Value Decomposition of the Complex Matrix," *Comm. ACM 12*, 564–65.

G.H. Golub and C. Reinsch (1970). "Singular Value Decomposition and Least Squares Solutions," *Numer. Math. 14*, 403–20. See also Wilkinson and Reinsch(1971, pp. 1334–51).

T.F. Chan (1982). "An Improved Algorithm for Computing the Singular Value Decomposition," *ACM Trans. Math. Soft. 8*, 72–83.

QR with column pivoting was first discussed in

P.A. Businger and G.H. Golub (1965). "Linear Least Squares Solutions by Householder Transformations," *Numer. Math.* 7, 269–76. See also Wilkinson and Reinsch (1971, pp. 11–18).

Knowing when to stop in the algorithm is difficult. In questions of rank deficiency, it is helpful to obtain information about the smallest singular value of the upper triangular matrix R. This can be done using the techniques of §3.5.4 or those that are discussed in

I. Karasalo (1974). "A Criterion for Truncation of the QR Decomposition Algorithm for the Singular Linear Least Squares Problem," *BIT 14*, 156–66.

N. Anderson and I. Karasalo (1975). "On Computing Bounds for the Least Singular Value of a Triangular Matrix," *BIT 15*, 1–4.

Other aspects of rank estimation with QR are discussed in

L.V. Foster (1986). "Rank and Null Space Calculations Using Matrix Decomposition without Column Interchanges," *Lin. Alg. and Its Applic. 74*, 47–71.

T.F. Chan (1987). "Rank Revealing QR Factorizations," *Lin. Alg. and Its Applic. 88/89*, 67–82.

T.F. Chan and P. Hansen (1992). "Some Applications of the Rank Revealing QR Factorization," *SIAM J. Sci. and Stat. Comp. 13*, 727–741.

J.L. Barlow and U.B. Vemulapati (1992). "Rank Detection Methods for Sparse Matrices," *SIAM J. Matrix. Anal. Appl. 13*, 1279–1297.

T-M. Hwang, W-W. Lin, and E.K. Yang (1992). "Rank-Revealing LU Factorizations," *Lin. Alg. and Its Applic. 175*, 115–141.

C.H. Bischof and P.C. Hansen (1992). "A Block Algorithm for Computing Rank-Revealing QR Factorizations," *Numerical Algorithms 2*, 371-392.

S. Chandrasekaren and I.C.F. Ipsen (1994). "On Rank-Revealing Factorizations," *SIAM J. Matrix Anal. Appl. 15*, 592–622.

R.D. Fierro and P.C. Hansen (1995). "Accuracy of TSVD Solutions Computed from Rank-Revealing Decompositions," *Numer. Math. 70*, 453–472.

5.5 The Rank Deficient LS Problem

If A is rank deficient, then there are an infinite number of solutions to the LS problem and we must resort to special techniques. These techniques must address the difficult problem of numerical rank determination.

After some SVD preliminaries, we show how QR with column pivoting can be used to determine a minimizer x_B with the property that Ax_B is a linear combination of $r = \text{rank}(A)$ columns. We then discuss the minimum 2-norm solution that can be obtained from the SVD.

5.5.1 The Minimum Norm Solution

Suppose $A \in \mathbb{R}^{m\times n}$ and $\text{rank}(A) = r < n$. The rank deficient LS problem has an infinite number of solutions, for if x is a minimizer and $z \in \text{null}(A)$ then $x + z$ is also a minimizer. The set of all minimizers

$$\mathcal{X} = \{x \in \mathbb{R}^n : \| Ax - b \|_2 = \min \}$$

is convex, for if $x_1, x_2 \in \mathcal{X}$ and $\lambda \in [0,1]$, then

$$\begin{aligned} \| A(\lambda x_1 + (1-\lambda)x_2) - b \|_2 &\le \lambda \| Ax_1 - b \|_2 + (1-\lambda) \| Ax_2 - b \|_2 \\ &= \min \| Ax - b \|_2 . \end{aligned}$$

Thus, $\lambda x_1 + (1-\lambda)x_2 \in \mathcal{X}$. It follows that $\mathcal{X}$ has a unique element having minimum 2-norm and we denote this solution by x_{LS}. (Note that in the full rank case, there is only one LS solution and so it must have minimal 2-norm. Thus, we are consistent with the notation in §5.3.)

5.5.2 Complete Orthogonal Factorization and x_{LS}

Any complete orthogonal factorization can be used to compute x_{LS}. In particular, if Q and Z are orthogonal matrices such that

$$Q^TAZ = T = \begin{bmatrix} T_{11} & 0 \\ 0 & 0 \end{bmatrix} \begin{matrix} r \\ m-r \end{matrix} \qquad r = \text{rank}(A)$$
$$\qquad\qquad\quad r \quad n-r$$

then

$$\| Ax - b \|_2^2 = \| (Q^TAZ)Z^Tx - Q^Tb \|_2^2 = \| T_{11}w - c \|_2^2 + \| d \|_2^2$$

where

$$Z^Tx = \begin{bmatrix} w \\ y \end{bmatrix} \begin{matrix} r \\ n-r \end{matrix} \qquad Q^Tb = \begin{bmatrix} c \\ d \end{bmatrix} \begin{matrix} r \\ m-r \end{matrix} .$$

Clearly, if x is to minimize the sum of squares, then we must have $w = T_{11}^{-1}c$. For x to have minimal 2-norm, y must be zero, and thus,

$$x_{LS} = Z \begin{bmatrix} T_{11}^{-1}c \\ 0 \end{bmatrix} .$$

5.5.3 The SVD and the LS Problem

Of course, the SVD is a particularly revealing complete orthogonal decomposition. It provides a neat expression for x_{LS} and the norm of the minimum residual $\rho_{LS} = \| Ax_{LS} - b \|_2$.

Theorem 5.5.1 *Suppose $U^TAV = \Sigma$ is the SVD of $A \in \mathbb{R}^{m\times n}$ with $r = \text{rank}(A)$. If $U = [\, u_1, \ldots, u_m \,]$ and $V = [\, v_1, \ldots, v_n \,]$ are column partitionings and $b \in \mathbb{R}^m$, then*

$$x_{LS} = \sum_{i=1}^{r} \frac{u_i^T b}{\sigma_i} v_i \tag{5.5.1}$$

minimizes $\| Ax - b \|_2$ and has the smallest 2-norm of all minimizers. Moreover

$$\rho_{LS}^2 = \| Ax_{LS} - b \|_2^2 = \sum_{i=r+1}^{m} (u_i^T b)^2. \tag{5.5.2}$$

Proof. For any $x \in \mathbb{R}^n$ we have:

$$\begin{aligned} \| Ax - b \|_2^2 &= \| (U^TAV)(V^Tx) - U^Tb \|_2^2 = \| \Sigma\alpha - U^Tb \|_2^2 \\ &= \sum_{i=1}^{r} (\sigma_i\alpha_i - u_i^Tb)^2 + \sum_{i=r+1}^{m} (u_i^Tb)^2 \end{aligned}$$

where $\alpha = V^Tx$. Clearly, if x solves the LS problem, then $\alpha_i = (u_i^Tb/\sigma_i)$ for $i = 1{:}r$. If we set $\alpha(r+1{:}n) = 0$, then the resulting x clearly has minimal 2-norm. □

5.5.4 The Pseudo-Inverse

Note that if we define the matrix $A^+ \in \mathbb{R}^{n\times m}$ by $A^+ = V\Sigma^+U^T$ where

$$\Sigma^+ = \text{diag}\left(\frac{1}{\sigma_1}, \ldots, \frac{1}{\sigma_r}, 0, \ldots, 0\right) \in \mathbb{R}^{n\times m} \qquad r = \text{rank}(A)$$

then $x_{LS} = A^+b$ and $\rho_{LS} = \| (I - AA^+)b \|_2$. A^+ is referred to as the *pseudo-inverse* of A. It is the unique minimal Frobenius norm solution to the problem

$$\min_{X \in \mathbb{R}^{n\times m}} \| AX - I_m \|_F . \tag{5.5.3}$$

If $\text{rank}(A) = n$, then $A^+ = (A^TA)^{-1}A^T$, while if $m = n = \text{rank}(A)$, then $A^+ = A^{-1}$. Typically, A^+ is defined to be the unique matrix $X \in \mathbb{R}^{n\times m}$ that satisfies the four *Moore-Penrose conditions:*

$$\begin{array}{llll} \text{(i)} & AXA = A & \text{(iii)} & (AX)^T = AX \\ \text{(ii)} & XAX = X & \text{(iv)} & (XA)^T = XA. \end{array}$$

These conditions amount to the requirement that AA^+ and A^+A be orthogonal projections onto $\text{ran}(A)$ and $\text{ran}(A^T)$, respectively. Indeed, $AA^+ = U_1U_1^T$ where $U_1 = U(1{:}m, 1{:}r)$ and $A^+A = V_1V_1^T$ where $V_1 = V(1{:}n, 1{:}r)$.

5.5.5 Some Sensitivity Issues

In §5.3 we examined the sensitivity of the full rank LS problem. The behavior of x_{LS} in this situation is summarized in Theorem 5.3.1. If we drop the full rank assumptions then x_{LS} is not even a continuous function of the data and small changes in A and b can induce arbitrarily large changes in $x_{LS} = A^+b$. The easiest way to see this is to consider the behavior of the pseudo inverse. If A and δA are in $\mathbb{R}^{m\times n}$, then Wedin (1973) and Stewart (1975) show that

$$\| (A+\delta A)^+ - A^+ \|_F \le 2\| \delta A \|_F \max\left\{ \| A^+ \|_2^2 \ , \ \| (A+\delta A)^+ \|_2^2 \right\}.$$

This inequality is a generalization of Theorem 2.3.4 in which perturbations in the matrix inverse are bounded. However, unlike the square nonsingular case, the upper bound does not necessarily tend to zero as δA tends to zero. If

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \qquad \text{and} \qquad \delta A = \begin{bmatrix} 0 & 0 \\ 0 & \epsilon \\ 0 & 0 \end{bmatrix}$$

then

$$A^+ = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \text{and} \qquad (A+\delta A)^+ = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1/\epsilon & 0 \end{bmatrix}$$

and $\| A^+ - (A+\delta A)^+ \|_2 = 1/\epsilon$. The numerical determination of an LS minimizer in the presence of such discontinuities is a major challenge.

5.5.6 QR with Column Pivoting and Basic Solutions

Suppose $A \in \mathbb{R}^{m\times n}$ has rank r. QR with column pivoting (Algorithm 5.4.1) produces the factorization $A\Pi = QR$ where

$$R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix} \begin{array}{l} r \\ m-r \end{array} .$$
$$\qquad\quad r \quad n-r$$

Given this reduction, the LS problem can be readily solved. Indeed, for any $x \in \mathbb{R}^n$ we have

$$\begin{aligned} \| Ax - b \|_2^2 &= \| (Q^TA\Pi)(\Pi^Tx) - (Q^Tb) \|_2^2 \\ &= \| R_{11}y - (c - R_{12}z) \|_2^2 + \| d \|_2^2 , \end{aligned}$$

where

$$\Pi^T x = \begin{bmatrix} y \\ z \end{bmatrix} \begin{matrix} r \\ n-r \end{matrix} \quad \text{and} \quad Q^T b = \begin{bmatrix} c \\ d \end{bmatrix} \begin{matrix} r \\ m-r \end{matrix} .$$

Thus, if x is an LS minimizer, then we must have

$$x = \Pi \begin{bmatrix} R_{11}^{-1}(c - R_{12}z) \\ z \end{bmatrix} .$$

If z is set to zero in this expression, then we obtain the *basic solution*

$$x_B = \Pi \begin{bmatrix} R_{11}^{-1}c \\ 0 \end{bmatrix} .$$

Notice that x_B has at most r nonzero components and so Ax_B involves a subset of A's columns.

The basic solution is not the minimal 2-norm solution unless the submatrix R_{12} is zero since

$$\| x_{LS} \|_2 = \min_{z \in \mathbb{R}^{n-r}} \left\| x_B - \Pi \begin{bmatrix} R_{11}^{-1}R_{12} \\ -I_{n-r} \end{bmatrix} z \right\|_2 . \qquad (5.5.4)$$

Indeed, this characterization of $\| x_{LS} \|_2$ can be used to show

$$1 \le \frac{\| x_B \|_2}{\| x_{LS} \|_2} \le \sqrt{1 + \| R_{11}^{-1}R_{12} \|_2^2} . \qquad (5.5.5)$$

See Golub and Pereyra (1976) for details.

5.5.7 Numerical Rank Determination with $A\Pi = QR$

If Algorithm 5.4.1 is used to compute x_B, then care must be exercised in the determination of rank(A). In order to appreciate the difficulty of this, suppose

$$fl(H_k \cdots H_1 A \Pi_1 \cdots \Pi_k) = \hat{R}^{(k)} = \begin{bmatrix} \hat{R}_{11}^{(k)} & \hat{R}_{12}^{(k)} \\ 0 & \hat{R}_{22}^{(k)} \end{bmatrix} \begin{matrix} k \\ m-k \end{matrix}$$
$$\qquad\qquad\qquad\qquad\qquad\qquad k \quad n-k$$

is the matrix computed after k steps of the algorithm have been executed in floating point. Suppose rank(A) = k. Because of roundoff error, $\hat{R}_{22}^{(k)}$ will not be exactly zero. However, if $\hat{R}_{22}^{(k)}$ is suitably small in norm then it is reasonable to terminate the reduction and declare A to have rank k. A typical termination criteria might be

$$\| \hat{R}_{22}^{(k)} \|_2 \le \epsilon_1 \| A \|_2 \qquad (5.5.6)$$

for some small machine-dependent parameter ϵ_1. In view of the roundoff properties associated with Householder matrix computation (cf. §5.1.12), we know that $\hat{R}^{(k)}$ is the exact R factor of a matrix $A + E_k$, where

$$\| E_k \|_2 \le \epsilon_2 \| A \|_2 \qquad \epsilon_2 = O(\mathbf{u}) .$$

Using Theorem 2.5.2 we have

$$\sigma_{k+1}(A + E_k) = \sigma_{k+1}(\hat{R}^{(k)}) \le \| \hat{R}_{22}^{(k)} \|_2 .$$

Since $\sigma_{k+1}(A) \le \sigma_{k+1}(A + E_k) + \| E_k \|_2$, it follows that

$$\sigma_{k+1}(A) \le (\epsilon_1 + \epsilon_2) \| A \|_2 .$$

In other words, a relative perturbation of $O(\epsilon_1 + \epsilon_2)$ in A can yield a rank-k matrix. With this termination criterion, we conclude that QR with column pivoting "discovers" rank degeneracy if in the course of the reduction $\hat{R}_{22}^{(k)}$ is small for some $k < n$.

Unfortunately, this is not always the case. A matrix can be nearly rank deficient without a single $\hat{R}_{22}^{(k)}$ being particularly small. Thus, QR with column pivoting *by itself* is not entirely reliable as a method for detecting near rank deficiency. However, if a good condition estimator is applied to R it is practically impossible for near rank deficiency to go unnoticed.

Example 5.5.1 Let $T_n(c)$ be the matrix

$$T_n(c) = \text{diag}(1, s, \ldots, s^{n-1}) \begin{bmatrix} 1 & -c & -c & \cdots & -c \\ 0 & 1 & -c & \cdots & -c \\ & \ddots & & \vdots & \vdots \\ \vdots & & & 1 & -c \\ 0 & & \cdots & & 1 \end{bmatrix}$$

with $c^2 + s^2 = 1$ with $c, s > 0$ (See Lawson and Hanson (1974, p.31).) These matrices are unaltered by Algorithm 5.4.1 and thus $\| R_{22}^{(k)} \|_2 \ge s^{n-1}$ for $k = 1{:}n-1$. This inequality implies (for example) that the matrix $T_{100}(.2)$ has no particularly small trailing principal submatrix since $s^{99} \approx .13$. However, it can be shown that $\sigma_n = O(10^{-8})$.

5.5.8 Numerical Rank and the SVD

We now focus our attention on the ability of the SVD to handle rank-deficiency in the presence of roundoff. Recall that if $A = U\Sigma V^T$ is the SVD of A, then

$$x_{LS} = \sum_{i=1}^{r} \frac{u_i^T b}{\sigma_i} v_i \qquad (5.5.7)$$

where r = rank(A). Denote the computed versions of U, V, and Σ = diag(σ_i) by $\hat{U}$, $\hat{V}$, and $\hat{\Sigma}$ = diag($\hat{\sigma}_i$). Assume that both sequences of singular

values range from largest to smallest. For a reasonably implemented SVD algorithm it can be shown that

$$\hat{U} = W + \Delta U \qquad W^TW = I_m \qquad \| \Delta U \|_2 \leq \epsilon \tag{5.5.8}$$

$$\hat{V} = Z + \Delta V \qquad Z^TZ = I_n \qquad \| \Delta V \|_2 \leq \epsilon \tag{5.5.9}$$

$$\hat{\Sigma} = W^T(A + \Delta A)Z \qquad \| \Delta A \|_2 \leq \epsilon \| A \|_2 \tag{5.5.10}$$

where ϵ is a small multiple of $\mathbf{u}$, the machine precision. In plain English, the SVD algorithm computes the singular values of a "nearby" matrix $A + \Delta A$.

Note that $\hat{U}$ and $\hat{V}$ are not necessarily close to their exact counterparts. However, we can show that $\hat{\sigma}_k$ is close to σ_k. Using (5.5.10) and Theorem 2.5.2 we have

$$\sigma_k = \min_{\text{rank}(B)=k-1} \| A - B \|_2$$

$$= \min_{\text{rank}(B)=k-1} \| (\hat{\Sigma} - B) - W^T(\Delta A)Z \|_2 .$$

Since $\| W^T(\Delta A)Z \|_2 \leq \epsilon \| A \|_2 = \epsilon\sigma_1$ and

$$\min_{\text{rank}(B)=k-1} \| \hat{\Sigma}_k - B \|_2 = \hat{\sigma}_k$$

it follows that $|\sigma_k - \hat{\sigma}_k| \leq \epsilon\sigma_1$ for $k = 1{:}n$. Thus, if A has rank r then we can expect $n - r$ of the computed singular values to be small. Near rank deficiency in A cannot escape detection when the SVD of A is computed.

Example 5.5.2 For the matrix $T_{100}(.2)$ in Example 5.5.1, $\sigma_n \approx .367 \cdot 10^{-8}$.

One approach to estimating $r = \text{rank}(A)$ from the computed singular values is to have a tolerance $\delta > 0$ and a convention that A has "numerical rank" $\hat{r}$ if the $\hat{\sigma}_i$ satisfy

$$\hat{\sigma}_1 \geq \cdots \geq \hat{\sigma}_{\hat{r}} > \delta \geq \hat{\sigma}_{\hat{r}+1} \geq \cdots \geq \hat{\sigma}_n .$$

The tolerance δ should be consistent with the machine precision, e.g. $\delta = \mathbf{u}\| A \|_\infty$. However, if the general level of relative error in the data is larger than $\mathbf{u}$, then δ should be correspondingly bigger, e.g., $\delta = 10^{-2}\| A \|_\infty$ if the entries in A are correct to two digits.

If $\hat{r}$ is accepted as the numerical rank then we can regard

$$x_{\hat{r}} = \sum_{i=1}^{\hat{r}} \frac{\hat{u}_i^T b}{\hat{\sigma}_i} \hat{v}_i$$

as an approximation to x_{LS}. Since $\| x_{\hat{r}} \|_2 \approx 1/\sigma_{\hat{r}} \leq 1/\delta$ then δ may also be chosen with the intention of producing an approximate LS solution with suitably small norm. In §12.1, we discuss more sophisticated methods for doing this.

If $\hat{\sigma}_{\hat{r}} \gg \delta$, then we have reason to be comfortable with $x_{\hat{r}}$ because A can then be unambiguously regarded as a $\text{rank}(A_{\hat{r}})$ matrix (modulo δ).

On the other hand, $\{\hat{\sigma}_1, \ldots, \hat{\sigma}_n\}$ might not clearly split into subsets of small and large singular values, making the determination of $\hat{r}$ by this means somewhat arbitrary. This leads to more complicated methods for estimating rank which we now discuss in the context of the LS problem.

For example, suppose $r = n$, and assume for the moment that $\Delta A = 0$ in (5.5.10). Thus $\sigma_i = \hat{\sigma}_i$ for $i = 1{:}n$. Denote the ith columns of the matrices $\hat{U}$, $\hat{W}$, $\hat{V}$, and Z by u_i, w_i, v_i, and z_i, respectively. Subtracting $x_{\hat{r}}$ from x_{LS} and taking norms we obtain

$$\| x_{\hat{r}} - x_{LS} \|_2 \leq \sum_{i=1}^{\hat{r}} \frac{\| (w_i^Tb)z_i - (u_i^Tb)v_i \|_2}{\sigma_i} + \sqrt{\sum_{i=\hat{r}+1}^{n} \left(\frac{w_i^Tb}{\sigma_i}\right)^2} .$$

From (5.5.8) and (5.5.9) it is easy to verify that

$$\| (w_i^Tb)z_i - (u_i^Tb)v_i \|_2 \leq 2(1+\epsilon)\epsilon \| b \|_2 \tag{5.5.11}$$

and therefore

$$\| x_{\hat{r}} - x_{LS} \|_2 \leq \frac{\hat{r}}{\sigma_{\hat{r}}} 2(1+\epsilon)\epsilon \| b \|_2 + \sqrt{\sum_{i=\hat{r}+1}^{n} \left(\frac{w_i^Tb}{\sigma_i}\right)^2} .$$

The parameter $\hat{r}$ can be determined as that integer which minimizes the upper bound. Notice that the first term in the bound increases with $\hat{r}$, while the second decreases.

On occasions when minimizing the residual is more important than accuracy in the solution, we can determine $\hat{r}$ on the basis of how close we surmise $\| b - Ax_{\hat{r}} \|_2$ is to the true minimum. Paralleling the above analysis, it can be shown that

$$\| b - Ax_{\hat{r}} \|_2 - \| b - Ax_{LS} \|_2 \leq (n - \hat{r})\| b \|_2 + \epsilon \| b \|_2 \left(\hat{r} + \frac{\hat{\sigma}_1}{\hat{\sigma}_{\hat{r}}}(1+\epsilon)\right) .$$

Again $\hat{r}$ could be chosen to minimize the upper bound. See Varah (1973) for practical details and also the LAPACK manual.

5.5.9 Some Comparisons

As we mentioned, when solving the LS problem via the SVD, only Σ and V have to be computed. The following table compares the efficiency of this approach with the other algorithms that we have presented.

LS Algorithm	Flop Count
Normal Equations	$mn^2 + n^3/3$
Householder Orthogonalization	$2mn^2 - 2n^3/3$
Modified Gram Schmidt	$2mn^2$
Givens Orthogonalization	$3mn^2 - n^3$
Householder Bidiagonalization	$4mn^2 - 4n^3/2$
R-Bidiagonalization	$2mn^2 + 2n^3$
Golub-Reinsch SVD	$4mn^2 + 8n^3$
R-SVD	$2mn^2 + 11n^3$

Problems

P5.5.1 Show that if

$$A = \begin{array}{cc} \begin{bmatrix} T & S \\ 0 & 0 \end{bmatrix} & \begin{matrix} r \\ m-r \end{matrix} \\ \begin{matrix} r & n-r \end{matrix} & \end{array}$$

where $r = \text{rank}(A)$ and T is nonsingular, then

$$X = \begin{array}{cc} \begin{bmatrix} T^{-1} & 0 \\ 0 & 0 \end{bmatrix} & \begin{matrix} r \\ n-r \end{matrix} \\ \begin{matrix} r & m-r \end{matrix} & \end{array}$$

satisfies $AXA = A$ and $(AX)^T = (AX)$. In this case, we say that X is a (1,3) *pseudo-inverse* of A. Show that for general A, $x_B = Xb$ where X is a (1,3) pseudo-inverse of A.

P5.5.2 Define $B(\lambda) \in \mathbb{R}^{n\times m}$ by $B(\lambda) = (A^TA + \lambda I)^{-1}A^T$, where $\lambda > 0$. Show

$$\| B(\lambda) - A^+ \|_2 = \frac{\lambda}{\sigma_r(A)[\sigma_r(A)^2 + \lambda]} \qquad r = \text{rank}(A)$$

and therefore that $B(\lambda) \to A^+$ as $\lambda \to 0$.

P5.5.3 Consider the rank deficient LS problem

$$\min_{\substack{y\in\mathbb{R}^r \\ z\in\mathbb{R}^{n-r}}} \left\| \begin{bmatrix} R & S \\ 0 & 0 \end{bmatrix} \begin{bmatrix} y \\ z \end{bmatrix} - \begin{bmatrix} c \\ d \end{bmatrix} \right\|_2$$

where $R \in \mathbb{R}^{r\times r}$, $S \in \mathbb{R}^{r\times n-r}$, $y \in \mathbb{R}^r$, and $z \in \mathbb{R}^{n-r}$. Assume that R is upper triangular and nonsingular. Show how to obtain the minimum norm solution to this problem by computing an appropriate QR factorization without pivoting and then solving for the appropriate y and z.

P5.5.4 Show that if $A_k \to A$ and $A_k^+ \to A^+$, then there exists an integer k_0 such that $\text{rank}(A_k)$ is constant for all $k \geq k_0$.

P5.5.5 Show that if $A \in \mathbb{R}^{m\times n}$ has rank n, then so does $A+E$ if we have the inequality $\| E \|_2 \| A^+ \|_2 < 1$.

Notes and References for Sec. 5.5

The pseudo-inverse literature is vast, as evidenced by the 1,775 references in

M.Z. Nashed (1976). *Generalized Inverses and Applications*, Academic Press, New York.

The differentiation of the pseudo-inverse is further discussed in

C.L. Lawson and R.J. Hanson (1969). "Extensions and Applications of the Householder Algorithm for Solving Linear Least Squares Problems," *Math. Comp. 23*, 787–812.
G.H. Golub and V. Pereyra (1973). "The Differentiation of Pseudo-Inverses and Nonlinear Least Squares Problems Whose Variables Separate," *SIAM J. Num. Anal. 10*, 413–32.

Survey treatments of LS perturbation theory may be found in Lawson and Hanson (1974), Stewart and Sun (1991), Björck (1996), and

P.A. Wedin (1973). "Perturbation Theory for Pseudo-Inverses," *BIT 13*, 217–32.
G.W. Stewart (1977). "On the Perturbation of Pseudo-Inverses, Projections, and Linear Least Squares," *SIAM Review 19*, 634–62.

Even for full rank problems, column pivoting seems to produce more accurate solutions. The error analysis in the following paper attempts to explain why.

L.S. Jennings and M.R. Osborne (1974). "A Direct Error Analysis for Least Squares," *Numer. Math. 22*, 322–32.

Various other aspects rank deficiency are discussed in

J.M. Varah (1973). "On the Numerical Solution of Ill-Conditioned Linear Systems with Applications to Ill-Posed Problems," *SIAM J. Num. Anal. 10*, 257–67.
G.W. Stewart (1984). "Rank Degeneracy," *SIAM J. Sci. and Stat. Comp. 5*, 403–413.
P.C. Hansen (1987). "The Truncated SVD as a Method for Regularization," *BIT 27*, 534–553.
G.W. Stewart (1987). "Collinearity and Least Squares Regression," *Statistical Science 2*, 68–100.

We have more to say on the subject in §12.1 and §12.2.

5.6 Weighting and Iterative Improvement

The concepts of scaling and iterative improvement were introduced in the Chapter 3 context of square linear systems. Generalizations of these ideas that are applicable to the least squares problem are now offered.

5.6.1 Column Weighting

Suppose $G \in \mathbb{R}^{n\times n}$ is nonsingular. A solution to the LS problem

$$\min \| Ax - b \|_2 \qquad A \in \mathbb{R}^{m\times n},\ b \in \mathbb{R}^m \tag{5.6.1}$$

can be obtained by finding the minimum 2-norm solution y_{LS} to

$$\min \| (AG)y - b \|_2 \tag{5.6.2}$$

and then setting $x_G = Gy_{LS}$. If $\text{rank}(A) = n$, then $x_G = x_{LS}$. Otherwise, x_G is the minimum G-norm solution to (5.6.1), where the G-norm is defined by $\| z \|_G = \| G^{-1}z \|_2$.

The choice of G is important. Sometimes its selection can be based on à priori knowledge of the uncertainties in A. On other occasions, it may be desirable to normalize the columns of A by setting

$$G \;=\; G_0 \;\equiv\; \text{diag}(1/\| A(:,1) \|_2, \ldots, 1/\| A(:,n) \|_2)\,.$$

Van der Sluis (1969) has shown that with this choice, $\kappa_2(AG)$ is approximately minimized. Since the computed accuracy of y_{LS} depends on $\kappa_2(AG)$, a case can be made for setting $G = G_0$.

We remark that column weighting affects singular values. Consequently, a scheme for determining numerical rank may not return the same estimates when applied to A and AG. See Stewart (1984b).

5.6.2 Row Weighting

Let $D = \text{diag}(d_1, \ldots, d_m)$ be nonsingular and consider the *weighted least squares problem*

$$\text{minimize } \| D(Ax - b) \|_2 \qquad A \in \mathbb{R}^{m\times n},\ b \in \mathbb{R}^m\,. \tag{5.6.3}$$

Assume $\text{rank}(A) = n$ and that x_D solves (5.6.3). It follows that the solution x_{LS} to (5.6.1) satisfies

$$x_D - x_{LS} \;=\; (A^TD^2A)^{-1}A^T(D^2 - I)(b - Ax_{LS})\,. \tag{5.6.4}$$

This shows that row weighting in the LS problem affects the solution. (An important exception occurs when $b \in \text{ran}(A)$ for then $x_D = x_{LS}$.)

One way of determining D is to let d_k be some measure of the uncertainty in b_k, e.g., the reciprocal of the standard deviation in b_k. The tendency is for $r_k = e_k^T(b - Ax_D)$ to be small whenever d_k is large. The precise effect of d_k on r_k can be clarified as follows. Define

$$D(\delta) \;=\; \text{diag}(d_1, \ldots, d_{k-1},\, d_k\sqrt{1+\delta}\,, d_{k+1}, \ldots, d_m)$$

where $\delta > -1$. If $x(\delta)$ minimizes $\| D(\delta)(Ax - b) \|_2$ and $r_k(\delta)$ is the k-th component of $b - Ax(\delta)$, then it can be shown that

$$r_k(\delta) \;=\; \frac{r_k}{1 + \delta d_k^2 e_k^T A(A^TD^2A)^{-1}A^Te_k}\,. \tag{5.6.5}$$

This explicit expression shows that $r_k(\delta)$ is a monotone decreasing function of δ. Of course, how r_k changes when all the weights are varied is much more complicated.

Example 5.6.1 Suppose

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix} \qquad b = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

If $D = I_4$ then $x_D = [\,-1,\ .85\,]^T$ and $r = b - Ax_D = [\,.3,\ -.4,\ -.1,\ .2\,]^T$. On the other hand, if $D = \text{diag}(\,1000,\ 1,\ 1,\ 1\,)$ then we have $x_D \approx [\,-1.43,\ 1.21\,]^T$ and $r = b - Ax_D = [\,.000428\ -.571428\ -.142853\ .285714\,]^T$.

5.6.3 Generalized Least Squares

In many estimation problems, the vector of observations b is related to x through the equation

$$b \;=\; Ax + w \tag{5.6.6}$$

where the *noise vector* w has zero mean and a symmetric positive definite *variance-covariance* matrix σ^2W. Assume that W is known and that $W = BB^T$ for some $B \in \mathbb{R}^{m\times m}$. The matrix B might be given or it might be W's Cholesky triangle. In order that all the equations in (5.6.6) contribute equally to the determination of x, statisticians frequently solve the LS problem

$$\min \| B^{-1}(Ax - b) \|_2\,. \tag{5.6.7}$$

An obvious computational approach to this problem is to form $\tilde{A} = B^{-1}A$ and $\tilde{b} = B^{-1}b$ and then apply any of our previous techniques to minimize $\| \tilde{A}x - \tilde{b} \|_2$. Unfortunately, x will be poorly determined by such a procedure if B is ill-conditioned.

A much more stable way of solving (5.6.7) using orthogonal transformations has been suggested by Paige (1979a, 1979b). It is based on the idea that (5.6.7) is equivalent to the *generalized least squares* problem,

$$\min_{b = Ax + Bv} \; v^Tv\,. \tag{5.6.8}$$

Notice that this problem is defined even if A and B are rank deficient. Although Paige's technique can be applied when this is the case, we shall describe it under the assumption that both these matrices have full rank.

The first step is to compute the QR factorization of A:

$$Q^TA \;=\; \begin{bmatrix} R_1 \\ 0 \end{bmatrix} \qquad Q \;=\; [\, \underset{n}{Q_1} \quad \underset{m-n}{Q_2} \,]\,.$$

An orthogonal matrix $Z \in \mathbb{R}^{m\times m}$ is then determined so that

$$Q_2^TBZ \;=\; [\, \underset{n}{0} \quad \underset{m-n}{S} \,] \qquad Z \;=\; [\, \underset{n}{Z_1} \quad \underset{m-n}{Z_2} \,]$$

where S is upper triangular. With the use of these orthogonal matrices the constraint in (5.6.8) transforms to

$$\begin{bmatrix} Q_1^Tb \\ Q_2^Tb \end{bmatrix} \;=\; \begin{bmatrix} R_1 \\ 0 \end{bmatrix} x \;+\; \begin{bmatrix} Q_1^TBZ_1 & Q_1^TBZ_2 \\ 0 & S \end{bmatrix} \begin{bmatrix} Z_1^Tv \\ Z_2^Tv \end{bmatrix}.$$

Notice that the "bottom half" of this equation determines v,

$$Su = Q_2^T b \qquad v = Z_2 u, \tag{5.6.9}$$

while the "top half" prescribes x:

$$R_1 x = Q_1^T b - (Q_1^T B Z_1 Z_1^T + Q_1^T B Z_2 Z_2^T)v = Q_1^T b - Q_1^T B Z_2 u. \tag{5.6.10}$$

The attractiveness of this method is that all potential ill-conditioning is concentrated in triangular systems (5.6.9) and (5.6.10). Moreover, Paige (1979b) has shown that the above procedure is numerically stable, something that is not true of any method that explicitly forms $B^{-1}A$.

5.6.4 Iterative Improvement

A technique for refining an approximate LS solution has been analyzed by Björck (1967, 1968). It is based on the idea that if

$$\begin{bmatrix} I_m & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} r \\ x \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix} \qquad A \in \mathbb{R}^{m\times n},\ b \in \mathbb{R}^m \tag{5.6.11}$$

then $\| b - Ax \|_2 = \min$. This follows because $r + Ax = b$ and $A^T r = 0$ imply $A^T A x = A^T b$. The above augmented system is nonsingular if $\text{rank}(A) = n$, which we hereafter assume.

By casting the LS problem in the form of a square linear system, the iterative improvement scheme (3.5.5) can be applied:

$r^{(0)} = 0;\ x^{(0)} = 0$
for $k = 0, 1, \ldots$

$$\begin{bmatrix} f^{(k)} \\ g^{(k)} \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix} - \begin{bmatrix} I & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} r^{(k)} \\ x^{(k)} \end{bmatrix}$$

$$\begin{bmatrix} I & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} p^{(k)} \\ z^{(k)} \end{bmatrix} = \begin{bmatrix} f^{(k)} \\ g^{(k)} \end{bmatrix}$$

$$\begin{bmatrix} r^{(k+1)} \\ x^{(k+1)} \end{bmatrix} = \begin{bmatrix} r^{(k)} \\ x^{(k)} \end{bmatrix} + \begin{bmatrix} p^{(k)} \\ z^{(k)} \end{bmatrix}$$

end

The residuals $f^{(k)}$ and $g^{(k)}$ must be computed in higher precision and an original copy of A must be around for this purpose.

If the QR factorization of A is available, then the solution of the augmented system is readily obtained. In particular, if $A = QR$ and $R_1 = R(1{:}n, 1{:}n)$, then a system of the form

$$\begin{bmatrix} I & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} p \\ z \end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix}$$

transforms to

$$\begin{bmatrix} I_n & 0 & R_1 \\ 0 & I_{m-n} & 0 \\ R_1^T & 0 & 0 \end{bmatrix} \begin{bmatrix} h \\ f_2 \\ z \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ g \end{bmatrix}$$

where

$$Q^T f = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} \begin{matrix} n \\ m-n \end{matrix} \qquad Q^T p = \begin{bmatrix} h \\ f_2 \end{bmatrix} \begin{matrix} n \\ m-n \end{matrix}.$$

Thus, p and z can be determined by solving the triangular systems $R_1^T h = g$ and $R_1 z = f_1 - h$ and setting $p = Q \begin{bmatrix} h \\ f_2 \end{bmatrix}$. Assuming that Q is stored in factored form, each iteration requires $8mn - 2n^2$ flops.

The key to the iteration's success is that both the LS residual and solution are updated—not just the solution. Björck (1968) shows that if $\kappa_2(A) \approx \beta^q$ and t-digit, β-base arithmetic is used, then $x^{(k)}$ has approximately $k(t-q)$ correct base β digits, provided the residuals are computed in double precision. Notice that it is $\kappa_2(A)$, not $\kappa_2(A)^2$, that appears in this heuristic.

Problems

P5.6.1 Verify (5.6.4).

P5.6.2 Let $A \in \mathbb{R}^{m\times n}$ have full rank and define the diagonal matrix

$$\Delta = \text{diag}(\underbrace{1,\ldots,1}_{k-1}, (1+\delta), \underbrace{1,\ldots,1}_{m-k})$$

for $\delta > -1$. Denote the LS solution to $\min \| \Delta(Ax - b) \|_2$ by $x(\delta)$ and its residual by $r(\delta) = b - Ax(\delta)$. (a) Show

$$r(\delta) = \left(I - \delta \frac{A(A^TA)^{-1}A^T e_k e_k^T}{1 + \delta e_k^T A(A^TA)^{-1}A^T e_k} \right) r(0).$$

(b) Letting $r_k(\delta)$ stand for the kth component of $r(\delta)$, show

$$r_k(\delta) = \frac{r_k(0)}{1 + \delta e_k^T A(A^TA)^{-1}A^T e_k}.$$

(c) Use (b) to verify (5.6.5).

P5.6.3 Show how the SVD can be used to solve the generalized LS problem when the matrices A and B in (5.6.8) are rank deficient.

P5.6.4 Let $A \in \mathbb{R}^{m \times n}$ have rank n and for $\alpha \geq 0$ define

$$M(\alpha) = \begin{bmatrix} \alpha I_m & A \\ A^T & 0 \end{bmatrix}.$$

Show that

$$\sigma_{m+n}(M(\alpha)) = \min\left\{\alpha\ ,\ -\frac{\alpha}{2} + \sqrt{\sigma_n(A)^2 + \left(\frac{\alpha}{2}\right)^2}\right\}$$

and determine the value of α that minimizes $\kappa_2(M(\alpha))$.

P5.6.5 Another iterative improvement method for LS problems is the following:

$x^{(0)} = 0$
for $k = 0, 1, \ldots$
  $r^{(k)} = b - Ax^{(k)}$ (double precision)
  $\| Az^{(k)} - r^{(k)} \|_2 = \min$
  $x^{(k+1)} = x^{(k)} + z^{(k)}$
end

(a) Assuming that the QR factorization of A is available, how many flops per iteration are required? (b) Show that the above iteration results by setting $g^{(k)} = 0$ in the iterative improvement scheme given in §5.6.4.

Notes and References for Sec. 5.6

Row and column weighting in the LS problem is discussed in Lawson and Hanson (SLS, pp. 180-88). The various effects of scaling are discussed in

A. van der Sluis (1969). "Condition Numbers and Equilibration of Matrices," *Numer. Math. 14*, 14–23.

G.W. Stewart (1984b). "On the Asymptotic Behavior of Scaled Singular Value and QR Decompositions," *Math. Comp. 43*, 483–490.

The theoretical and computational aspects of the generalized least squares problem appear in

S. Kourouklis and C.C. Paige (1981). "A Constrained Least Squares Approach to the General Gauss-Markov Linear Model," *J. Amer. Stat. Assoc. 76*, 620-25.

C.C. Paige (1979a). "Computer Solution and Perturbation Analysis of Generalized Least Squares Problems," *Math. Comp. 33*, 171–84.

C.C. Paige (1979b). "Fast Numerically Stable Computations for Generalized Linear Least Squares Problems," *SIAM J. Num. Anal. 16*, 165–71.

C.C. Paige (1985). "The General Limit Model and the Generalized Singular Value Decomposition," *Lin. Alg. and Its Applic. 70*, 269–284.

Iterative improvement in the least squares context is discussed in

G.H. Golub and J.H. Wilkinson (1966). "Note on Iterative Refinement of Least Squares Solutions," *Numer. Math. 9*, 139–48.

Å. Björck and G.H. Golub (1967). "Iterative Refinement of Linear Least Squares Solutions by Householder Transformation," *BIT 7*, 322-37.

Å. Björck (1967). "Iterative Refinement of Linear Least Squares Solutions I," *BIT 7*, 257–78.

Å. Björck (1968). "Iterative Refinement of Linear Least Squares Solutions II," *BIT 8*, 8–30.

Å. Björck (1987). "Stability Analysis of the Method of Seminormal Equations for Linear Least Squares Problems," *Linear Alg. and Its Applic. 88/89*, 31–48.

5.7 Square and Underdetermined Systems

The orthogonalization methods developed in this chapter can be applied to square systems and also to systems in which there are fewer equations than unknowns. In this brief section we discuss some of the various possibilities.

5.7.1 Using QR and SVD to Solve Square Systems

The least squares solvers based on the QR factorization and the SVD can be used to solve square linear systems: just set $m = n$. However, from the flop point of view, Gaussian elimination is the cheapest way to solve a square linear system as shown in the following table which assumes that the right hand side is available at the time of factorization:

Method	Flops
Gaussian Elimination	$2n^3/3$
Householder Orthogonalization	$4n^3/3$
Modified Gram-Schmidt	$2n^3$
Bidiagonalization	$8n^3/3$
Singular Value Decomposition	$12n^3$

Nevertheless, there are three reasons why orthogonalization methods might be considered:

- The flop counts tend to exaggerate the Gaussian elimination advantage. When memory traffic and vectorization overheads are considered, the QR approach is comparable in efficiency.
- The orthogonalization methods have guaranteed stability; there is no "growth factor" to worry about as in Gaussian elimination.
- In cases of ill-conditioning, the orthogonal methods give an added measure of reliability. QR with condition estimation is very dependable and, of course, SVD is unsurpassed when it comes to producing a meaningful solution to a nearly singular system.

We are not expressing a strong preference for orthogonalization methods but merely suggesting viable alternatives to Gaussian elimination.

We also mention that the SVD entry in Table 5.7.1 assumes the availability of b at the time of decomposition. Otherwise, $20n^3$ flops are required because it then becomes necessary to accumulate the U matrix.

If the QR factorization is used to solve $Ax = b$, then we ordinarily have to carry out a back substitution: $Rx = Q^Tb$. However, this can be avoided by "preprocessing" b. Suppose H is a Householder matrix such

that $Hb = \beta e_n$ where e_n is the last column of I_n. If we compute the QR factorization of $(HA)^T$, then $A = H^T R^T Q^T$ and the system transforms to

$$R^T y = \beta e_n$$

where $y = Q^T x$. Since R^T is lower triangular, $y = (\beta/r_{nn})e_n$ and so

$$x = \frac{\beta}{r_{nn}} Q(:,n).$$

5.7.2 Underdetermined Systems

We say that a linear system

$$Ax = b \qquad A \in \mathbb{R}^{m\times n},\ b \in \mathbb{R}^m \tag{5.7.1}$$

is *underdetermined* whenever $m < n$. Notice that such a system either has no solution or has an infinity of solutions. In the second case, it is important to distinguish between algorithms that find the minimum 2-norm solution and those that do not necessarily do so. The first algorithm we present is in the latter category. Assume that A has full row rank and that we apply QR with column pivoting to obtain:

$$Q^T A \Pi = [\, R_1 \ R_2 \,]$$

where $R_1 \in \mathbb{R}^{m\times m}$ is upper triangular and $R_2 \in \mathbb{R}^{m\times(n-m)}$. Thus, $Ax = b$ transforms to

$$(Q^T A \Pi)(\Pi^T x) = [\, R_1 \ R_2 \,] \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = Q^T b$$

where

$$\Pi^T x = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$$

with $z_1 \in \mathbb{R}^m$ and $z_2 \in \mathbb{R}^{(n-m)}$. By virtue of the column pivoting, R_1 is nonsingular because we are assuming that A has full row rank. One solution to the problem is therefore obtained by setting $z_1 = R_1^{-1} Q^T b$ and $z_2 = 0$.

Algorithm 5.7.1 Given $A \in \mathbb{R}^{m\times n}$ with $\text{rank}(A) = m$ and $b \in \mathbb{R}^m$, the following algorithm finds an $x \in \mathbb{R}^n$ such that $Ax = b$.

$Q^T A \Pi = R$ (QR with column pivoting.)
Solve $R(1{:}m, 1{:}m) z_1 = Q^T b$.
Set $x = \Pi \begin{bmatrix} z_1 \\ 0 \end{bmatrix}$.

This algorithm requires $2m^2n - m^3/3$ flops. The minimum norm solution is not guaranteed. (A different Π would render a smaller z_1.) However, if we compute the QR factorization

$$A^T = QR = Q \begin{bmatrix} R_1 \\ 0 \end{bmatrix}$$

with $R_1 \in \mathbb{R}^{m\times m}$, then $Ax = b$ becomes

$$(QR)^T x = [\, R_1^T \ \ 0 \,] \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = b$$

where

$$Q^T x = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \qquad z_1 \in \mathbb{R}^m,\ z_2 \in \mathbb{R}^{n-m}.$$

Now the minimum norm solution *does* follow by setting $z_2 = 0$.

Algorithm 5.7.2 Given $A \in \mathbb{R}^{m\times n}$ with $\text{rank}(A) = m$ and $b \in \mathbb{R}^m$, the following algorithm finds the minimal 2-norm solution to $Ax = b$.

$A^T = QR$ (QR factorization)
Solve $R(1{:}m, 1{:}m)^T z = b$.
$x = Q(:, 1{:}m) z$

This algorithm requires at most $2m^2n - 2m^3/3$

The SVD can also be used to compute the minimal norm solution of an underdetermined $Ax = b$ problem. If

$$A = \sum_{i=1}^{r} \sigma_i u_i v_i^T \qquad r = \text{rank}(A)$$

is A's singular value expansion, then

$$x = \sum_{i=1}^{r} \frac{u_i^T b}{\sigma_i} v_i\,.$$

As in the least squares problem, the SVD approach is desirable whenever A is nearly rank deficient.

5.7.3 Perturbed Underdetermined Systems

We conclude this section with a perturbation result for full-rank underdetermined systems.

Theorem 5.7.1 *Suppose* $\text{rank}(A) = m \le n$ *and that* $A \in \mathbb{R}^{m\times n}$, $\delta A \in \mathbb{R}^{m\times n}$, $0 \ne b \in \mathbb{R}^m$, *and* $\delta b \in \mathbb{R}^m$ *satisfy*

$$\epsilon = \max\{\epsilon_A, \epsilon_b\} < \sigma_m(A),$$

where $\epsilon_A = \| \delta A \|_2/\| A \|_2$ *and* $\epsilon_b = \| \delta b \|_2/\| b \|_2$. *If* x *and* $\hat{x}$ *are minimum norm solutions that satisfy*

$$Ax = b \qquad\qquad (A + \delta A)\hat{x} = b + \delta b$$

then

$$\frac{\| \hat{x} - x \|_2}{\| x \|_2} \le \kappa_2(A)\left(\epsilon_A \min\{2, n-m+1\} + \epsilon_b\right) + O(\epsilon^2).$$

Proof. Let E and f be defined by $\delta A/\epsilon$ and $\delta b/\epsilon$. Note that $\text{rank}(A + tE) = m$ for all $0 < t < \epsilon$ and that

$$x(t) = (A + tE)^T\left((A+tE)(A+tE)^T\right)^{-1}(b + tf)$$

satisfies $(A + tE)x(t) = b + tf$. By differentiating this expression with respect to t and setting $t = 0$ in the result we obtain

$$\dot{x}(0) = \left(I - A^T(AA^T)^{-1}A\right)E^T(AA^T)^{-1}b + A^T(AA^T)^{-1}(f - Ex).$$

Since

$$\| x \|_2 = \| A^T(AA^T)^{-1}b \|_2 \ge \sigma_m(A)\| (AA^T)^{-1}b \|_2,$$

$$\| I - A^T(AA^T)^{-1}A \|_2 = \min(1, n-m),$$

and

$$\frac{\| f \|_2}{\| x \|_2} \le \frac{\| f \|_2 \| A \|_2}{\| b \|_2},$$

we have

$$\frac{\| \hat{x} - x \|_2}{\| x \|_2} = \frac{x(\epsilon) - x(0)}{\| x(0) \|_2} = \epsilon\frac{\| \dot{x}(0) \|_2}{\| x \|_2} + O(\epsilon^2)$$

$$\le \epsilon \min(1, n-m)\left\{\frac{\| E \|_2}{\| A \|_2} + \frac{\| f \|_2}{\| b \|_2} + \frac{\| E \|_2}{\| A \|_2}\right\}\kappa_2(A) + O(\epsilon^2)$$

from which the theorem follows. $\square$

Note that there is no $\kappa_2(A)^2$ factor as in the case of overdetermined systems.

Problems

P5.7.1 Derive the above expression for $\dot{x}(0)$.

P5.7.2 Find the minimal norm solution to the system $Ax = b$ where $A = [\,1\ 2\ 3\,]$ and $b = 1$.

P5.7.3 Show how triangular system solving can be avoided when using the QR factorization to solve an underdetermined system.

P5.7.4 Suppose $b, x \in \mathbb{R}^n$ are given. Consider the following problems:

(a) Find an unsymmetric Toeplitz matrix T so $Tx = b$.

(b) Find a symmetric Toeplitz matrix T so $Tx = b$.

(c) Find a circulant matrix C so $Cx = b$.

Pose each problem in the form $Ap = b$ where A is a matrix made up of entries from x and p is the vector of sought-after parameters.

Notes and References for Sec. 5.7

Interesting aspects concerning singular systems are discussed in

T.F. Chan (1984). "Deflated Decomposition Solutions of Nearly Singular Systems," *SIAM J. Num. Anal.* *21*, 738–754.

G.H. Golub and C.D. Meyer (1986). "Using the QR Factorization and Group Inversion to Compute, Differentiate, and estimate the Sensitivity of Stationary Probabilities for Markov Chains," *SIAM J. Alg. and Dis. Methods*, 7, 273–281.

Papers on underdetermined systems include

R.E. Cline and R.J. Plemmons (1976). "L_2-Solutions to Underdetermined Linear Systems," *SIAM Review 18*, 92-106.

M. Arioli and A. Laratta (1985). "Error Analysis of an Algorithm for Solving an Underdetermined System," *Numer. Math. 46*, 255–268.

J.W. Demmel and N.J. Higham (1993). "Improved Error Bounds for Underdetermined System Solvers," *SIAM J. Matrix Anal. Appl. 14*, 1–14.

The QR factorization can of course be used to solve linear systems. See

N.J. Higham (1991). "Iterative Refinement Enhances the Stability of QR Factorization Methods for Solving Linear Equations," *BIT 31*, 447–468.

Chapter 6

Parallel Matrix Computations

§6.1 Basic Concepts
§6.2 Matrix Multiplication
§6.3 Factorizations

The parallel matrix computation area has been the focus of intense research. Although much of the work is machine/system dependent, a number of basic strategies have emerged. Our aim is to present these along with a picture of what it is like to "think parallel" during the design of a matrix computation.

The distributed and shared memory paradigms are considered. We use matrix-vector multiplication to introduce the notion of a node program in §6.1. Load balancing, speed-up, and synchronization are also discussed. In §6.2 matrix-matrix multiplication is used to show the effect of blocking on granularity and to convey the spirit of two-dimensional data flow. Two parallel implementations of the Cholesky factorization are given in §6.3.

Before You Begin

Chapter 1, §4.1, and §4.2 are assumed. Within this chapter there are the following dependencies:

$$\S6.1 \rightarrow \S6.2 \rightarrow \S6.3$$

Complementary references include the books by Schönauer (1987), Hockney and Jesshope (1988), Modi (1988), Ortega (1988), Dongarra, Duff,

Sorensen, and van der Vorst (1991), and Golub and Ortega (1993) and the excellent review papers by Heller (1978), Ortega and Voight (1985), Gallivan, Plemmons, and Sameh (1990), and Demmel, Heath, and van der Vorst (1993).

6.1 Basic Concepts

In this section we introduce the distributed and shared memory paradigms using the gaxpy operation

$$z = y + Ax, \qquad A \in \mathbb{R}^{n\times n}, x, y, z \in \mathbb{R}^n \tag{6.1.1}$$

as an example. In practice, there is a fuzzy line between these two styles of parallel computing and typically a blend of our comments apply to any particular machine.

6.1.1 Distributed Memory Systems

In a distributed memory multiprocessor each processor has a *local memory* and executes its own *node program*. The program can alter values in the executing processor's local memory and can send data in the form of *messages* to the other processors in the network. The interconnection of the processors defines the *network topology* and one simple example that is good enough for our introduction is the *ring*. See FIGURE 6.1.1. Other

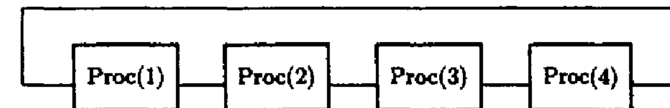


FIGURE 6.1.1 *A Four-Processor Ring*

important interconnection schemes include the mesh and torus (for their close correspondence with two-dimensional arrays), the hypercube (for its generality and optimality), and the tree (for its handling of divide and conquer procedures). See Ortega and Voigt (1985) for a discussion of the possibilities. Our immediate goal is to develop a ring algorithm for (6.1.1). Matrix multiplication on a torus is discussed in §6.2.

Each processor has an *identification number*. The μth processor is designated by Proc(μ). We say that Proc(λ) is a *neighbor* of Proc(μ) if there is a direct physical connection between them. Thus, in a p-processor ring, Proc($p-1$) and Proc(1) are neighbors of Proc(p).

Important factors in the design of an effective distributed memory algorithm include (a) the number of processors and the capacity of the local memories, (b) how the processors are interconnected, (c) the speed of computation relative to the speed of interprocessor communication, and (d) whether or not a node is able to compute and communicate at the same time.

6.1.2 Communication

To describe the sending and receiving of messages we adopt a simple notation:

send({*matrix*} , {id of the receiving processor})

recv({*matrix*} , {id of the sending processor})

Scalars and vectors are matrices and therefore messages. In our model, if Proc(μ) executes the instruction send(V_{loc}, λ), then a copy of the local matrix V_{loc} is sent to Proc(λ) and the execution of Proc(μ)'s node program resumes immediately. It is legal for a processor to send a message to itself. To emphasize that a matrix is stored in a local memory we use the subscript "*loc*."

If Proc(μ) executes the instruction recv(U_{loc}, λ), then the execution of its node program is suspended until a message is received from Proc(λ). Once received, the message is placed in a local matrix U_{loc} and Proc(μ) resumes execution of its node program.

Although the syntax and semantics of our send/receive notation is adequate for our purposes, it does suppress a number of important details:

- Message assembly overhead. In practice, there may be a penalty associated with the transmission of a matrix whose entries are not contiguous in the sender's local memory. We ignore this detail.

- Message tagging. Messages need not arrive in the order they are sent, and a system of message tagging is necessary so that the receiver is not "confused." We ignore this detail by assuming that messages *do* arrive in the order that they are sent.

- Message interpretation overhead. In practice a message is a bit string, and a header must be provided that indicates to the receiver the dimensions of the matrix and the format of the floating point words that are used to represent its entries. Going from message to stored matrix takes time, but it is an overhead that we do not try to quantify.

These simplifications enable us to focus on high-level algorithmic ideas. But it should be remembered that the success of a particular implementation may hinge upon the control of these hidden overheads.

6.1.3 Some Distributed Data Structures

Before we can specify our first distributed memory algorithm, we must consider the matter of *data layout*. How are the participating matrices and vectors distributed around the network?

Suppose $x \in \mathbb{R}^n$ is to be distributed among the local memories of a p-processor network. Assume for the moment that $n = rp$. Two "canonical" approaches to this problem are store-by-row and store-by-column.

In store-by-column we regard the vector x as an r-by-p matrix,

$$x_{r\times p} = \left[\; x(1{:}r) \quad x(r+1{:}2r) \quad \cdots \quad x(1+(p-1)r{:}n) \;\right],$$

and store each column in a processor, i.e., $x(1+(\mu-1)r{:}\mu r) \in \text{Proc}(\mu)$. (In this context "$\in$" means "is stored in.") Note that each processor houses a contiguous portion of x.

In the store-by-row scheme we regard x as a p-by-r matrix

$$x_{p\times r} = \left[\; x(1{:}p) \quad x(p+1{:}2p) \quad \cdots \quad x((r-1)p+1{:}n) \;\right],$$

and store each *row* in a processor, i.e., $x(\mu{:}p{:}n) \in \text{Proc}(\mu)$. Store-by-row is sometimes referred to as the *wrap* method of distributing a vector because the components of x can be thought of as cards in a deck that are "dealt" to the processors in wrap-around fashion.

If n is not an exact multiple of p, then these ideas go through with minor modification. Consider store-by-column with $n = 14$ and $p = 4$:

$$x^T = [\underbrace{x_1\, x_2\, x_3\, x_4}_{\text{Proc(1)}} \mid \underbrace{x_5\, x_6\, x_7\, x_8}_{\text{Proc(2)}} \mid \underbrace{x_9\, x_{10}\, x_{11}}_{\text{Proc(3)}} \mid \underbrace{x_{12}\, x_{13}\, x_{14}}_{\text{Proc(4)}}].$$

In general, if $n = pr + q$ with $0 \le q < p$, then Proc(1),...,Proc(q) can each house $r+1$ components and Proc($q+1$),..., Proc(p) can house r components. In store-by-row we simply let Proc(μ) house $x(\mu{:}p{:}n)$.

Similar options apply to the layout of a matrix. There are four obvious possibilities if $A \in \mathbb{R}^{n\times n}$ and (for simplicity) $n = rp$:

Orientation	Style	What is in Proc(μ)
Column	Contiguous	$A(:,1+(\mu-1)r{:}\mu r)$
Column	Wrap	$A(:,\mu{:}p{:}n)$
Row	Contiguous	$A(1+(\mu-1)r{:}\mu r,:)$
Row	Wrap	$A(\mu{:}p{:}n,:)$

These strategies have block analogs. For example, if $A = [A_1,\ldots,A_N]$ is a block column partitioning, then we could arrange to have Proc(μ) store A_i for $i = \mu{:}p{:}N$.

6.1.4 Gaxpy on a Ring

We are now set to develop a ring algorithm for the gaxpy $z = y + Ax$ ($A \in \mathbb{R}^{n\times n}$, $x, y \in \mathbb{R}^n$). For clarity, assume that $n = rp$ where p is the size of the ring. Partition the gaxpy as

$$\begin{bmatrix} z_1 \\ \vdots \\ z_p \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_p \end{bmatrix} + \begin{bmatrix} A_{11} & \cdots & A_{1p} \\ \vdots & & \vdots \\ A_{p1} & \cdots & A_{pp} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_p \end{bmatrix}. \qquad (6.1.2)$$

where $A_{ij} \in \mathbb{R}^{r\times r}$ and $x_i, y_i, z_i \in \mathbb{R}^r$. We assume that at the start of computation Proc(μ) houses x_μ, y_μ, and the μth block row of A. Upon completion we set as our goal the overwriting of y_μ by z_μ. From the Proc(μ) perspective, the computation of

$$z_\mu = y_\mu + \sum_{\tau=1}^{p} A_{\mu\tau} x_\tau$$

involves local data ($A_{\mu\tau}, y_\mu, x_\mu$) and nonlocal data ($x_\tau$, $\tau \neq \mu$). To make the nonlocal portions of x available, we circulate its subvectors around the ring. For example, in the $p = 3$ case we rotate the x_1, x_2, and x_3 as follows:

step	Proc(1)	Proc(2)	Proc(3)
1	x_3	x_1	x_2
2	x_2	x_3	x_1
3	x_1	x_2	x_3

When a subvector of x "visits", the host processor must incorporate the appropriate term into its running sum:

step	Proc(1)	Proc(2)	Proc(3)
1	$y_1 = y_1 + A_{13}x_3$	$y_2 = y_2 + A_{21}x_1$	$y_3 = y_3 + A_{32}x_2$
2	$y_1 = y_1 + A_{12}x_2$	$y_2 = y_2 + A_{23}x_3$	$y_3 = y_3 + A_{31}x_1$
3	$y_1 = y_1 + A_{11}x_1$	$y_2 = y_2 + A_{22}x_2$	$y_3 = y_3 + A_{33}x_3$

In general, the "merry-go-round" of x subvectors makes p "stops." For each received x-subvector, a processor performs an r-by-r gaxpy.

Algorithm 6.1.1 Suppose $A \in \mathbb{R}^{n\times n}$, $x \in \mathbb{R}^n$, and $y \in \mathbb{R}^n$ are given and that $z = y + Ax$. If each processor in a p-processor ring executes the following node program and $n = rp$, then upon completion Proc(μ) houses $z(1+(\mu-1)r:\mu r)$ in y_{loc}. Assume the following local memory initializations: p, μ (the node id), *left* and *right* (the neighbor id's), n, $row = 1+(\mu-1)r:\mu r$, $A_{loc} = A(row,:)$, $x_{loc} = x(row)$, $y_{loc} = y(row)$.


```
for t = 1:p
    send(x_loc, right)
    recv(x_loc, left)
    τ = μ − t
    if τ ≤ 0
        τ = τ + p
    end
    { x_loc = x(1 + (τ − 1)r:τr) }
    y_loc = y_loc + A_loc(:, 1 + (τ − 1)r:τr)x_loc
end
```

The index τ names the currently available x subvector. Once it is computed it is possible to carry out the update of the locally housed portion of y. The **send-recv** pair passes the currently housed x subvector to the right and waits to receive the next one from the left. Synchronization is achieved because the local y update cannot begin until the "new" x subvector arrives. It is impossible for one processor to "race ahead" of the others or for an x subvector to pass another in the merry-go-round. The algorithm is tailored to the ring topology in that only nearest neighbor communication is involved. The computation is also perfectly *load balanced* meaning that each processor has the same amount of computation and communication. Load imbalance is discussed further in §6.1.7.

The design of a parallel program involves subtleties that do not arise in the uniprocessor setting. For example, if we inadvertently reverse the order of the **send** and the **recv**, then each processor starts its node program by waiting for a message from its *left* neighbor. Since that neighbor in turn is waiting for a message from *its* left neighbor, a state of *deadlock* results.

6.1.5 The Cost of Communication

Communication overheads can be estimated if we model the cost of sending and receiving a message. To that end we assume that a **send** or **recv** involving m floating point numbers requires

$$\tau(m) = \alpha_d + \beta_d m \qquad (6.1.3)$$

seconds to carry out. Here α_d is the time required to initiate the **send** or **recv** and β_d is the reciprocal of the rate that a message can be transferred. Note that this model does not take into consideration the "distance" between the sender and receiver. Clearly, it takes longer to pass a message halfway around a ring than to a neighbor. That is why it is always desirable to arrange (if possible) a distributed computation so that communication is just between neighbors.

During each step in Algorithm 6.1.1 an r-vector is sent and received and $2r^2$ flops are performed. If the computation proceeds at R flops per second

and there is no idle waiting associated with the recv, then each y_{loc} update requires approximately $(2r^2/R) + 2(\alpha_d + \beta_d r)$ seconds.

Another instructive statistic is the *computation-to-communication ratio*. For Algorithm 6.1.1 this is prescribed by

$$\frac{\text{Time spent computing}}{\text{Time spent communicating}} \approx \frac{2r^2/R}{2(\alpha_d + \beta_d r)}.$$

This fraction quantifies the overhead of communication relative to the volume of computation. Clearly, as $r = n/p$ grows, the fraction of time spent computing increases.[1]

6.1.6 Efficiency and Speed-Up

The *efficiency* of a p-processor parallel algorithm is given by

$$E = \frac{T(1)}{pT(p)}$$

where $T(k)$ is the time required to execute the program on k processors. If computation proceeds at R flops/sec and communication is modeled by (6.1.3), then a reasonable estimate of $T(k)$ for Algorithm 6.1.1 is given by

$$T(k) = \sum_{i=1}^{k} 2(n/k)^2/R + 2(\alpha_d + \beta_d(n/k)) = \frac{2n^2}{Rk} + 2\alpha_d k + 2\beta_d n$$

for $k > 1$. This assumes no idle waiting. If $k = 1$, then no communication is required and $T(1) = 2n^2/R$. It follows that the efficiency

$$E = \frac{1}{1 + \frac{pR}{n}\left(\alpha_d \frac{p}{n} + \beta\right)}.$$

improves with increasing n and degradates with increasing p or R. In practice, benchmarking is the only dependable way to assess efficiency.

A concept related to efficiency is *speed-up*. We say that a parallel algorithm for a particular problem achieves speed-up S if

$$S = T_{seq}/T_{par}$$

where T_{par} is the time required for execution of the parallel program and T_{seq} is the time required by one processor when the best uniprocessor procedure is used. For some problems, the fastest sequential algorithm does not parallelize and so two distinct algorithms are involved in the speed-up assessment.

[1] We mention that these simple measures are not particularly illuminating in systems where the nodes are able to overlap computation and communication.

6.1.7 The Challenge of Load Balancing

If we apply Algorithm 6.1.1 to a matrix $A \in \mathbb{R}^{n\times n}$ that is lower triangular, then approximately half of the flops associated with the y_{loc} updates are unnecessary because half of the A_{ij} in (6.1.2) are zero. In particular, in the μth processor, $A_{loc}(:, 1 + (\tau - 1)r{:}\tau r)$ is zero if $\tau > \mu$. Thus, if we guard the y_{loc} update as follows,

```
if τ ≤ μ
    y_loc = y_loc + A_loc(:, 1 + (τ − 1)r:τr) x_loc
end
```

then the overall number of flops is halved. This solves the superfluous flops problem but it creates a load imbalance problem. Proc(μ) oversees about $\mu r^2/2$ flops, an increasing function of the processor id μ. Consider the following $r = p = 3$ example:

$$\begin{bmatrix} z_1\\ z_2\\ z_3\\ \hline z_4\\ z_5\\ z_6\\ \hline z_7\\ z_8\\ z_9 \end{bmatrix} = \left[\begin{array}{ccc|ccc|ccc} \alpha&0&0&0&0&0&0&0&0\\ \alpha&\alpha&0&0&0&0&0&0&0\\ \alpha&\alpha&\alpha&0&0&0&0&0&0\\ \hline \beta&\beta&\beta&\beta&0&0&0&0&0\\ \beta&\beta&\beta&\beta&\beta&0&0&0&0\\ \beta&\beta&\beta&\beta&\beta&\beta&0&0&0\\ \hline \gamma&\gamma&\gamma&\gamma&\gamma&\gamma&\gamma&0&0\\ \gamma&\gamma&\gamma&\gamma&\gamma&\gamma&\gamma&\gamma&0\\ \gamma&\gamma&\gamma&\gamma&\gamma&\gamma&\gamma&\gamma&\gamma \end{array}\right] \begin{bmatrix} x_1\\ x_2\\ x_3\\ \hline x_4\\ x_5\\ x_6\\ \hline x_7\\ x_8\\ x_9 \end{bmatrix} + \begin{bmatrix} y_1\\ y_2\\ y_3\\ \hline y_4\\ y_5\\ y_6\\ \hline y_7\\ y_8\\ y_9 \end{bmatrix}$$

Here, Proc(1) handles the α part, Proc(2) handles the β part, and Proc(3) handles the γ part.

However, if processors 1, 2, and 3 compute (z_1, z_4, z_7), (z_2, z_5, z_8), and (z_3, z_6, z_9), respectively, then approximate load balancing results:

$$\begin{bmatrix} z_1\\ z_4\\ z_7\\ \hline z_2\\ z_5\\ z_8\\ \hline z_3\\ z_6\\ z_9 \end{bmatrix} = \left[\begin{array}{ccc|ccc|ccc} \alpha&0&0&0&0&0&0&0&0\\ \beta&\beta&\beta&\beta&0&0&0&0&0\\ \gamma&\gamma&\gamma&\gamma&\gamma&\gamma&\gamma&0&0\\ \hline \alpha&\alpha&0&0&0&0&0&0&0\\ \beta&\beta&\beta&\beta&\beta&0&0&0&0\\ \gamma&\gamma&\gamma&\gamma&\gamma&\gamma&\gamma&\gamma&0\\ \hline \alpha&\alpha&\alpha&0&0&0&0&0&0\\ \beta&\beta&\beta&\beta&\beta&\beta&0&0&0\\ \gamma&\gamma&\gamma&\gamma&\gamma&\gamma&\gamma&\gamma&\gamma \end{array}\right] \begin{bmatrix} x_1\\ x_2\\ x_3\\ \hline x_4\\ x_5\\ x_6\\ \hline x_7\\ x_8\\ x_9 \end{bmatrix} + \begin{bmatrix} y_1\\ y_4\\ y_7\\ \hline y_2\\ y_5\\ y_8\\ \hline y_3\\ y_6\\ y_9 \end{bmatrix}$$

The amount of arithmetic still increases with μ, but the effect is not noticeable if $n \gg p$.

The development of the general algorithm requires some index manipulation. Assume that Proc(μ) is initialized with $A_{loc} = A(\mu{:}p{:}n, :)$ and

$y_{loc} = y(\mu{:}p{:}n)$, and assume that the contiguous x-subvectors circulate as before. If at some stage x_{loc} contains $x(1+(\tau-1)r{:}\tau r)$, then the update

$$y_{loc} = y_{loc} + A_{loc}(:, 1+(\tau-1)r{:}\tau r)x_{loc}$$

implements

$$y(\mu{:}p{:}n) = y(\mu{:}p{:}n) + A(\mu{:}p{:}n, 1+(\tau-1)r{:}\tau r)x(1+(\tau-1)r{:}\tau r).$$

To exploit the triangular structure of A in the y_{loc} computation, we express the gaxpy as a double loop:

```
for α = 1:r
    for β = 1:r
        y_loc(α) = y_loc(α) + A_loc(α, β + (τ − 1)r)x_loc(β)
    end
end
```

The A_{loc} reference refers to $A(\mu+(\alpha-1)p, \beta+(\tau-1)r)$ which is zero unless the column index is less than or equal to the row index. Abbreviating the inner loop range with this in mind we obtain

Algorithm 6.1.2 Suppose $A \in \mathbb{R}^{n\times n}$, $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^n$ are given and that $z = y + Ax$. Assume that $n = rp$ and that A is lower triangular. If each processor in a p-processor ring executes the following node program, then upon completion Proc(μ) houses $z(\mu{:}p{:}n)$ in y_{loc}. Assume the following local memory initializations: p, μ (the node id), *left* and *right* (the neighbor id's), n, $A_{loc} = A(\mu{:}p{:}n, :)$, $y_{loc} = y(\mu{:}p{:}n)$, and $x_{loc} = x(1+(\mu-1)r{:}\mu r)$.

```
r = n/p
for t = 1:p
    send(x_loc, right)
    recv(x_loc, left)
    τ = μ − t
    if τ ≤ 0
        τ = τ + p
    end
    {x_loc = x(1 + (τ − 1)r:τr)}
    for α = 1:r
        for β = 1:μ + (α − 1)p − (τ − 1)r
            y_loc(α) = y_loc(α) + A_loc(α, β + (τ − 1)r)x_loc(β)
        end
    end
end
```

Having to map indices back and forth between "node space" and "global space" is one aspect of distributed matrix computations that requires care and (hopefully) compiler assistance.

6.1.8 Tradeoffs

As we did in §1.1, let us develop a column-oriented gaxpy and anticipate its performance. With the block column partitioning

$$A = [A_1, \ldots, A_p] \qquad A_i \in \mathbb{R}^{n\times r},\ r = n/p$$

the gaxpy $z = y + Ax$ becomes

$$z = y + \sum_{\mu=1}^{p} A_\mu x_\mu$$

where $x_\mu = x(1+(\mu-1)r{:}\mu r)$. Assume that Proc($\mu$) contains A_μ and x_μ. Its contribution to the gaxpy is the product $A_\mu x_\mu$ and involves local data. However, these products must be summed. We assign this task to Proc(1) which we assume contains y. The strategy is thus for each processor to compute $A_\mu x_\mu$ and to send the result to Proc(1).

Algorithm 6.1.3 Suppose $A \in \mathbb{R}^{n\times n}$, $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^n$ are given and that $z = y + Ax$. If each processor in a p-processor network executes the following node program and $n = rp$, then upon completion Proc(1) houses z. Assume the following local memory initializations: p, μ (the node id), n, $x_{loc} = x(1+(\mu-1)r{:}\mu r)$, $A_{loc} = A(:, 1+(\mu-1)r{:}\mu r)$, and (in Proc(1) only) $y_{loc} = y$.

```
if μ = 1
    y_loc = y_loc + A_loc x_loc
    for t = 2:p
        recv(w_loc, t)
        y_loc = y_loc + w_loc
    end
else
    w_loc = A_loc x_loc
    send(w_loc, 1)
end
```

At first glance this seems to be much less attractive than the row-oriented Algorithm 6.1.1. The additional responsibilities of Proc(1) mean that it has more arithmetic to perform by a factor of about

$$\frac{2n^2/p + np}{2n^2/p} = 1 + \frac{p^2}{2n}$$

and more messages to process by a factor of about p. This imbalance becomes less critical if $n \gg p$ and the communication parameters α_d and β_d factors are small enough. Another possible mitigating factor is that Algorithm 6.1.3 manipulates length n vectors whereas Algorithm 6.1.1 works

with length n/p vectors. If the nodes are capable of vector arithmetic, then the longer vectors may raise the level of performance.

This brief comparison of Algorithms 6.1.1 and 6.1.3 reminds us once again that different implementations of the same computation can have very different performance characteristics.

6.1.9 Shared Memory Systems

We now discuss the gaxpy problem for a shared memory multiprocessor. In this environment each processor has access to a common, global memory as depicted in Figure 6.1.2. Communication between processors is achieved

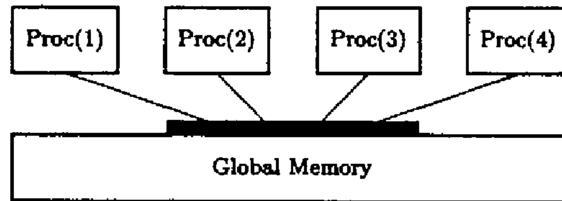


FIGURE 6.1.2 *A Four-Processor Shared Memory System*

by reading and writing to *global variables* that reside in the global memory. Each processor executes its own *local program* and has its own *local memory*. Data flows to and from the global memory during execution.

All the concerns that attend distributed memory computation are with us in modified form. The overall procedure should be *load balanced* and the computations should be arranged so that the individual processors have to wait as little as possible for something useful to compute. The traffic between the global and local memories must be managed carefully, because the extent of such data transfers is typically a significant overhead. (It corresponds to interprocessor communication in the distributed memory setting and to data motion up and down a memory hierarchy as discussed in §1.4.5.) The nature of the physical connection between the processors and the shared memory is very important and can effect algorithmic development. However, for simplicity we regard this aspect of the system as a black box as shown in Figure 6.1.2.

6.1.10 A Shared Memory Gaxpy

Consider the following partitioning of the n-by-n gaxpy problem $z = y + Ax$:

$$\begin{bmatrix} z_1 \\ \vdots \\ z_p \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_p \end{bmatrix} + \begin{bmatrix} A_1 \\ \vdots \\ A_p \end{bmatrix} x. \qquad (6.1.4)$$

Here we assume that $n = rp$ and that $A_\mu \in \mathbb{R}^{r \times n}$, $y_\mu \in \mathbb{R}^r$, and $z_\mu \in \mathbb{R}^r$. We use the following algorithm to introduce the basic ideas and notations.

Algorithm 6.1.4 Suppose $A \in \mathbb{R}^{n \times n}$, $x \in \mathbb{R}^n$, and $y \in \mathbb{R}^n$ reside in a global memory accessible to p processors. If $n = rp$ and each processor executes the following algorithm, then upon completion, y is overwritten by $z = y + Ax$. Assume the following initializations in each local memory: p, μ (the node id), and n.

$r = n/p$
$row = 1 + (\mu - 1)r{:}\mu r$
$x_{loc} = x$
$y_{loc} = y(row)$
for $j = 1{:}n$
  $a_{loc} = A(row, j)$
  $y_{loc} = y_{loc} + a_{loc} x_{loc}(j)$
end
$y(row) = y_{loc}$

We assume that a copy of this program resides in each processor. Floating point variables that are local to an individual processor have a "*loc*" subscript.

Data is transferred to and from the global memory during the execution of Algorithm 6.1.4. There are two global memory reads before the loop ($x_{loc} = x$ and $y_{loc} = y(row)$), one read each time through the loop ($a_{loc} = A(row, j)$), and one write after the loop ($y(row) = y_{loc}$).

Only one processor writes to a given global memory location in y, and so there is no need to synchronize the participating processors. Each has a completely independent part of the overall gaxpy operation and does not have to monitor the progress of the other processors. The computation is *statically scheduled* because the partitioning of work is determined before execution.

If A is lower triangular, then steps have to be taken to preserve the load balancing in Algorithm 6.1.4. As we discovered in §6.1.7, the wrap mapping is a vehicle for doing this. Assigning Proc(μ) the computation of $z(\mu{:}p{:}n) = y(\mu{:}p{:}n) + A(\mu{:}p{:}n, :)x$ effectively partitions the n^2 flops among the p processors.

6.1.11 Memory Traffic Overhead

It is important to recognize that overall performance depends strongly on the overheads associated with the reads and writes to the global memory. If such a data transfer involves m floating point numbers, then we model the transfer time by

$$\tau(m) = \alpha_s + \beta_s m. \tag{6.1.5}$$

The parameter α_s represents a start-up overhead and β_s is the reciprocal transfer rate. We modelled interprocessor communication in the distributed environment exactly the same way. (See (6.1.3).)

Accounting for all the shared memory reads and writes in Algorithm 6.1.4 we see that each processor spends time

$$T \approx (n+3)\alpha_s + \frac{n^2}{p}\beta_s$$

communicating with global memory.

We organized the computation so that one column of $A(row,:)$ is read at a time from shared memory. *If* the local memory is large enough, then the loop in Algorithm 6.1.4 can be replaced with

$$A_{loc} = A(row,:)$$
$$y_{loc} = y_{loc} + A_{loc}x_{loc}$$

This changes the communication overhead to

$$\bar{T} \approx 3\alpha_s + \frac{n^2}{p}\beta_s,$$

a significant improvement if the start-up parameter α_s is large.

6.1.12 Barrier Synchronization

Let us consider the shared memory version of Algorithm 6.1.4 in which the gaxpy is column oriented. Assume $n = rp$ and $col = 1 + (\mu - 1)r{:}\mu r$. A reasonable idea is to use a global array $W(1{:}n, 1{:}p)$ to house the products $A(:,col)x(col)$ produced by each processor, and then have some chosen processor (say Proc(1)) add its columns:

```
A_loc = A(:,col); x_loc = x(col); w_loc = A_loc x_loc; W(:,μ) = w_loc
if μ = 1
    y_loc = y
    for j = 1:p
        w_loc = W(:,j)
        y_loc = y_loc + w_loc
    end
    y = y_loc
end
```


However, this strategy is seriously flawed because there is no guarantee that $W(1{:}n, 1{:}p)$ is fully initialized when Proc(1) begins the summation process.

What we need is a synchronization construct that can delay the Proc(1) summation until all the processors have computed and stored their contributions in the W array. For this purpose many shared memory systems support some version of the **barrier** construct which we introduce in the following algorithm:

Algorithm 6.1.5 Suppose $A \in \mathbb{R}^{n\times n}$, $x \in \mathbb{R}^n$, and $y \in \mathbb{R}^n$ reside in a global memory accessible to p processors. If $n = rp$ and each processor executes the following algorithm, then upon completion y is overwritten by $y + Ax$. Assume the following initializations in each local memory: p, μ (the node id), and n.

```
r = n/p; col = 1 + (μ − 1)r:μr; A_loc = A(:,col); x_loc = x(col)
w_loc = A_loc x_loc
W(:,μ) = w_loc
barrier
if μ = 1
    y_loc = y
    for j = 1:p
        w_loc = W(:,j)
        y_loc = y_loc + w_loc
    end
    y = y_loc
end
```

To understand the **barrier**, it is convenient to regard a processor as either blocked or free. A processor is blocked and suspends execution when it executes the **barrier**. After the pth processor is blocked, all the processors return to the "free state" and resume execution. Think of the **barrier** as treacherous stream to be traversed by all p processors. For safety, they all congregate on the bank before attempting to cross. When the last member of the party arrives, they ford the stream in unison and resume their individual treks.

In Algorithm 6.1.5, the processors are blocked after computing their portion of the matrix-vector product. We cannot predict the order in which these blockings occur, but once the last processor reaches the **barrier**, they are all released and Proc(1) can carry out the vector summation.

6.1.13 Dynamic Scheduling

Instead of having one processor in charge of the vector summation, it is tempting to have each processor add its contribution directly to the global variable y. For Proc(μ), this means executing the following:

$r = n/p$; $col = 1 + (\mu - 1)r{:}\mu r$; $A_{loc} = A(:, col)$; $x_{loc} = x(col)$
$w_{loc} = A_{loc}x_{loc}$
$y_{loc} = y$; $y_{loc} = y_{loc} + w_{loc}$; $y = y_{loc}$

However, a problem concerns the read-update-write triplet

$y_{loc} = y$; $y_{loc} = y_{loc} + w_{loc}$; $y = y_{loc}$

Indeed, if more than one processor is executing this code fragment at the same time, then there may be a loss of information. Consider the following sequence:

Proc(1) reads y
Proc(2) reads y
Proc(1) writes y
Proc(2) writes y

The contribution of Proc(1) is lost because Proc(1) and Proc(2) obtain the same version of y. As a result, the effect of the Proc(1) write is erased by the Proc(2) write.

To prevent this kind of thing from happening most shared memory systems support the idea of a *critical section*. These are special, isolated portions of a node program that require a "key" to enter. Throughout the system, there is only one key and so the net effect is that only one processor can be executing in a critical section at any given time.

Algorithm 6.1.6 Suppose $A \in \mathbb{R}^{n \times n}$, $x \in \mathbb{R}^n$, and $y \in \mathbb{R}^n$ reside in a global memory accessible to p processors. If $n = pr$ and each processor executes the following algorithm, then upon completion, y is overwritten by $y + Ax$. Assume the following initializations in each local memory: p, μ (the node id), and n.

$r = n/p$; $col = 1 + (\mu - 1)r{:}\mu r$; $A_{loc} = A(:, col)$; $x_{loc} = x(col)$
$w_{loc} = A_{loc}x_{loc}$
begin critical section
    $y_{loc} = y$
    $y_{loc} = y_{loc} + w_{loc}$
    $y = y_{loc}$
end critical section

This use of the critical section concept controls the update of y in a way that ensures correctness. The algorithm is *dynamically scheduled* because the order in which the summations occur is determined as the computation unfolds. Dynamic scheduling is very important in problems with irregular structure.

Problems

P6.1.1 Modify Algorithm 6.1.1 so that it can handle arbitrary n.

P6.1.2 Modify Algorithm 6.1.2 so that it efficiently handles the upper triangular case.

P6.1.3 (a) Modify Algorithms 6.1.3 and 6.1.4 so that they overwrite y with $z = y + A^m x$ for a given positive integer m that is available to each processor. (b) Modify Algorithms 6.1.3 and 6.1.4 so that y is overwritten by $z = y + A^T Ax$.

P6.1.4 Modify Algorithm 6.1.3 so that upon completion, the local array A_{loc} in Proc(μ) houses the μth block column of $A + xy^T$.

P6.1.5 Modify Algorithm 6.1.4 so that (a) A is overwritten by the outer product update $A + xy^T$, (b) x is overwritten with A^2x, (c) y is overwritten by a unit 2-norm vector in the direction of $y + A^k x$, and (d) it efficiently handles the case when A is lower triangular.

Notes and References for Sec. 6.1

General references on parallel computations that include several chapters on matrix computations include

G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon and D. W. Walker (1988). *Solving Problems on Concurrent Processors, Volume 1*. Prentice Hall, Englewood Cliffs, NJ.

D. P. Bertsekas and J. N. Tsitsiklis (1989). *Parallel and Distributed Computation: Numerical Methods*, Prentice Hall, Englewood Cliffs, NJ.

S. Lakshmivarahan and S. K. Dhall (1990). *Analysis and Design of Parallel Algorithms: Arithmetic and Matrix Problems*, McGraw-Hill, New York.

T. L. Freeman and C. Phillips (1992). *Parallel Numerical Algorithms*, Prentice Hall, New York.

F.T. Leighton (1992). *Introduction to Parallel Algorithms and Architectures*, Morgan Kaufmann, San Mateo, CA.

G. C. Fox, R. D. Williams, and P. C. Messina (1994). *Parallel Computing Works!*, Morgan Kaufmann, San Francisco.

V. Kumar, A. Grama, A. Gupta and G. Karypis (1994). *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin/Cummings, Reading, MA.

E.F. Van de Velde (1994). *Concurrent Scientific Computing*, Springer-Verlag, New York.

M. Cosnard and D. Trystram (1995). *Parallel Algorithms and Architectures*, International Thomson Computer Press, New York.

Here are some general references that are more specific to parallel matrix computations:

V. Fadeeva and D. Fadeev (1977). "Parallel Computations in Linear Algebra," *Kibernetica 6*, 28–40.

D. Heller (1978). "A Survey of Parallel Algorithms in Numerical Linear Algebra," *SIAM Review 20*, 740-777.

J.M. Ortega and R.G. Voigt (1985). "Solution of Partial Differential Equations on Vector and Parallel Computers," *SIAM Review 27*, 149-240.

J.J. Dongarra and D.C. Sorensen (1986). "Linear Algebra on High Performance Computers," *Appl. Math. and Comp. 20*, 57–88.

K.A. Gallivan, R.J. Plemmons, and A.H. Sameh (1990). "Parallel Algorithms for Dense Linear Algebra Computations," *SIAM Review 32*, 54–135.

J.W. Demmel, M.T. Heath, and H.A. van der Vorst (1993) "Parallel Numerical Linear Algebra," in *Acta Numerica 1993*, Cambridge University Press.

See also

B.N. Datta (1989). "Parallel and Large-Scale Matrix Computations in Control: Some Ideas," *Lin. Alg. and Its Applic.* 121, 243–264.
A. Edelman (1993). "Large Dense Numerical Linear Algebra in 1993: The Parallel Computing Influence," *Int'l J. Supercomputer Appl.* 7, 113–128.

Managing and modelling communication in a distributed memory environment is an important, difficult problem. See

L. Adams and T. Crockett (1984). "Modeling Algorithm Execution Time on Processor Arrays," *Computer* 17, 38–43.
D. Gannon and J. Van Rosendale (1984). "On the Impact of Communication Complexity on the Design of Parallel Numerical Algorithms," *IEEE Trans. Comp.* C-33, 1180–1194.
S.L. Johnsson (1987). "Communication Efficient Basic Linear Algebra Computations on Hypercube Multiprocessors," *J. Parallel and Distributed Computing*, No. 4, 133–172.
Y. Saad and M. Schultz (1989). "Data Communication in Hypercubes," *J. Dist. Parallel Comp.* 6, 115–135.
Y. Saad and M.H. Schultz (1989). "Data Communication in Parallel Architectures," *J. Dist. Parallel Comp.* 11, 131–150.

For snapshots of basic linear algebra computation on a distributed memory system, see

O. McBryan and E.F. van de Velde (1987). "Hypercube Algorithms and Implementations," *SIAM J. Sci. and Stat. Comp.* 8, s227–s287.
S.L. Johnsson and C.T. Ho (1988). "Matrix Transposition on Boolean n-cube Configured Ensemble Architectures," *SIAM J. Matrix Anal. Appl.* 9, 419–454.
T. Dehn, M. Eiermann, K. Giebermann, and V. Sperling (1995). "Structured Sparse Matrix Vector Multiplication on Massively Parallel SIMD Architectures," *Parallel Computing* 21, 1867–1894.
J. Choi, J.J. Dongarra, and D.W. Walker (1995). "Parallel Matrix Transpose Algorithms on Distributed Memory Concurrent Computers," *Parallel Computing* 21, 1387–1406.
L. Colombet, Ph. Michallon, and D. Trystram (1996). "Parallel Matrix-Vector Product on Rings with a Minimum of Communication," *Parallel Computing* 22, 289–310.

The implementation of a parallel algorithm is usually very challenging. It is important to have compilers and related tools that are able to handle the details. See

D.P. O'Leary and G.W. Stewart (1986). "Assignment and Scheduling in Parallel Matrix Factorization," *Lin. Alg. and Its Applic.* 77, 275–300.
J. Dongarra and D.C. Sorensen (1987). "A Portable Environment for Developing Parallel Programs," *Parallel Computing* 5, 175–186.
K. Connolly, J.J. Dongarra, D. Sorensen, and J. Patterson (1988). "Programming Methodology and Performance Issues for Advanced Computer Architectures," *Parallel Computing* 5, 41–58.
P. Jacobson, B. Kagstrom, and M. Rannar (1992). "Algorithm Development for Distributed Memory Multicomputers Using Conlab," *Scientific Programming*, 1, 185–203.
C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell (1993). "A Linear Algebra Framework for Static HPF Code Distribution," *Proceedings of the 4th Workshop on Compilers for Parallel Computers*, Delft, The Netherlands.
D. Bau, I. Kodukula, V. Kotlyar, K. Pingali, and P. Stodghill (1993). "Solving Alignment Using Elementary Linear Algebra," in *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science 892, Springer-Verlag, New York, 46–60.
M. Wolfe (1996). *High Performance Compilers for Parallel Computers*, Addison-Wesley, Reading MA.

6.2 Matrix Multiplication

In this section we develop two parallel algorithms for matrix-matrix multiplication. A shared memory implementation is used to illustrate the effect of blocking on granularity and load balancing. A torus implementation is designed to convey the spirit of two-dimensional data flow.

6.2.1 A Block Gaxpy Procedure

Suppose $A, B, C \in \mathbb{R}^{n\times n}$ with B upper triangular and consider the computation of the matrix multiply update

$$D = C + AB \tag{6.2.1}$$

on a shared memory computer with p processors. Assume that $n = rkp$ and partition the update

$$[\, D_1, \ldots, D_{k\cdot p}\,] \;=\; [\, C_1, \ldots, C_{k\cdot p}\,] + [\, A_1, \ldots, A_{k\cdot p}\,][\, B_1, \ldots, B_{k\cdot p}\,] \tag{6.2.2}$$

where each block column has width $r = n/(kp)$. If

$$B_j \;=\; \begin{bmatrix} B_{1j} \\ \vdots \\ B_{jj} \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \qquad B_{ij} \in \mathbb{R}^{r\times r},$$

then

$$D_j \;=\; C_j + AB_j \;=\; C_j \;+\; \sum_{\tau=1}^{j} A_\tau B_{\tau j}. \tag{6.2.3}$$

The number of flops required to compute D_j is given by

$$f_j \;=\; 2nr^2 j \;=\; \left(\frac{2n^3}{k^2p^2}\right) j.$$

This is an increasing function of j because B is upper triangular. As we discovered in the previous section, the wrap mapping is the way to solve load imbalance problems that result from triangular matrix structure. This suggests that we assign Proc(μ) the task of computing D_j for $j = \mu{:}p{:}kp$.

Algorithm 6.2.1 Suppose A, B, and C are n-by-n matrices that reside in a global memory accessible to p processors. Assume that B is upper triangular and $n = rkp$. If each processor executes the following algorithm,

then upon completion C is overwritten by $D = C + AB$. Assume the following initializations in each local memory: n, r, k, p and μ (the node id).

```
for j = μ:p:kp
    {Compute D_j.}
    B_loc = B(1:jr, 1 + (j − 1)r:jr)
    C_loc = C(:, 1 + (j − 1)r:jr)
    for τ = 1:j
        col = 1 + (τ − 1)r:τr
        A_loc = A(:, col)
        C_loc = C_loc + A_loc B_loc(col, :)
    end
    C(:, 1 + (j − 1)r:jr) = C_loc
end
```

Let us examine the degree of load balancing as a function of the parameter k. For Proc(μ), the number of flops required is given by

$$F(\mu) = \sum_{i=1}^{k} f_{\mu+(i-1)p} \approx \left(k\mu + \frac{k^2p}{2}\right)\frac{2n^3}{k^2p^2}.$$

The quotient $F(p)/F(1)$ is a measure of load balancing from the flop point of view. Since

$$\frac{F(p)}{F(1)} = \frac{kp + k^2p/2}{k + k^2p/2} = 1 + \frac{2(p-1)}{2+kp}$$

we see that arithmetic balance improves with increasing k. A similar analysis shows that the communication overheads are well balanced as k increases.

On the other hand, the total number of global memory reads and writes associated with Algorithm 6.2.1 increases with the square of k. If the start-up parameter α_s in (6.1.5) is large, then performance can degrade with increased k.

The optimum choice for k given these two opposing forces is system dependent. If communication is fast, then smaller tasks can be supported without penalty and this makes it easier to achieve load balancing. A multiprocessor with this attribute supports *fine-grained parallelism*. However, if granularity is too fine in a system with high-performance nodes, then it may be impossible for the node programs to perform at level-2 or level-3 speeds simply because there just is not enough local linear algebra. Again, benchmarking is the only way to clarify these issues.

6.2.2 Torus

A torus is a two-dimensional processor array in which each row and column is a ring. See FIGURE 6.2.1. A Processor *id* in this context is an

ordered pair and each processor has four neighbors. In the displayed exam-

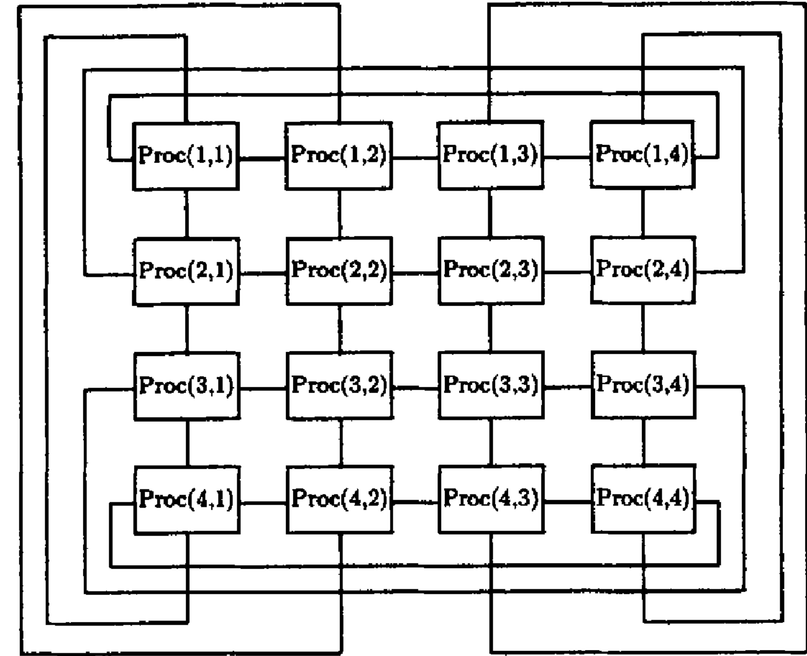


FIGURE 6.2.1 *A Four-by-Four Torus*

ple, Proc(1,3) has *west* neighbor Proc(1,2), *east* neighbor Proc(1,4), *south* neighbor Proc(2,3), and *north* neighbor Proc(4,3).

To show what it is like to organize a toroidal matrix computation, we develop an algorithm for the matrix multiplication $D = C + AB$ where $A, B, C \in \mathbb{R}^{n\times n}$. Assume that the torus is p_1-by-p_1 and that $n = rp_1$. Regard $A = (A_{ij})$, $B = (B_{ij})$, and $C = (C_{ij})$ as p_1-by-p_1 block matrices with r-by-r blocks. Assume that Proc(i,j) contains A_{ij}, B_{ij}, and C_{ij} and that its mission is to overwrite C_{ij} with

$$D_{ij} = C_{ij} + \sum_{k=1}^{p_1} A_{ik}B_{kj}.$$

We develop the general algorithm from the $p_1 = 3$ case, displaying the torus in cellular form as follows:

Proc(1,1)	Proc(1,2)	Proc(1,3)
Proc(2,1)	Proc(2,2)	Proc(2,3)
Proc(3,1)	Proc(3,2)	Proc(3,3)

Let us focus attention on Proc(1,1) and the calculation of

$$D_{11} = C_{11} + A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31}\,.$$

Suppose the six inputs that define this block dot product are positioned within the torus as follows:

A_{11} B_{11}	A_{12} ·	A_{13} ·
· B_{21}	· ·	· ·
· B_{31}	· ·	· ·

(Pay no attention to the "dots." They are later replaced by various A_{ij} and B_{ij}).

Our plan is to "ratchet" the first block row of A and the first block column of B through Proc(1,1) in a coordinated fashion. The pairs A_{11} and B_{11}, A_{12} and B_{21}, and A_{13} and B_{31} meet, are multiplied, and added into a running sum array C_{loc}:

A_{12} B_{21}	A_{13} ·	A_{11} ·
· B_{31}	· ·	· ·
· B_{11}	· ·	· ·

$$C_{loc} = C_{loc} + A_{12}B_{21}$$

A_{13} B_{31}	A_{11} ·	A_{12} ·
· B_{11}	· ·	· ·
· B_{21}	· ·	· ·

$$C_{loc} = C_{loc} + A_{13}B_{31}$$

A_{11} B_{11}	A_{12} ·	A_{13} ·
· B_{21}	· ·	· ·
· B_{31}	· ·	· ·

$$C_{loc} = C_{loc} + A_{11}B_{11}$$

Thus, after three steps, the local array C_{loc} in Proc(1,1) houses D_{11}.

We have organized the flow of data so that the A_{1j} migrate westwards and the B_{i1} migrate northwards through the torus. It is thus apparent that Proc(1,1) must execute a node program of the form:

```
for t = 1:3
    send(A_loc, west)
    send(B_loc, north)
    recv(A_loc, east)
    recv(B_loc, south)
    C_loc = C_loc + A_loc B_loc
end
```

The send-recv-send-recv sequence

```
for t = 1:3
    send(A_loc, west)
    recv(A_loc, east)
    send(B_loc, north)
    recv(B_loc, south)
    C_loc = C_loc + A_loc B_loc
end
```

also works. However, this induces unnecessary delays into the process because the B submatrix is not sent until the new A submatrix arrives.

We next consider the activity in Proc(1,2), Proc(1,3), Proc(2,1), and Proc(3,1). At this point in the development, these processors merely help circulate blocks A_{11}, A_{12}, and A_{13} and B_{11}, B_{21}, and B_{31}, respectively. If B_{32}, B_{12}, and B_{22} flowed through Proc(1,2) during these steps, then

$$D_{12} = C_{12} + A_{13}B_{32} + A_{11}B_{12} + A_{12}B_{22}$$

could be formed. Likewise, Proc(1,3) could compute

$$D_{13} = C_{13} + A_{11}B_{13} + A_{12}B_{23} + A_{13}B_{33}$$

if B_{13}, B_{23}, and B_{33} are available during $t = 1{:}3$. To this end we initialize the torus as follows

A_{11} B_{11}	A_{12} B_{22}	A_{13} B_{33}
· B_{21}	· B_{32}	· B_{13}
· B_{31}	· B_{12}	· B_{23}

With northward flow of the B_{ij} we get

A_{12} B_{21}	A_{13} B_{32}	A_{11} B_{13}
· B_{31}	· B_{12}	· B_{23}
· B_{11}	· B_{22}	· B_{33}

$t = 1$

A_{13} B_{31}	A_{11} B_{12}	a_{12} B_{23}
· B_{11}	· B_{22}	· B_{33}
· B_{21}	· B_{32}	· B_{13}

$t = 2$

A_{11} B_{11}	A_{12} B_{22}	a_{13} B_{33}
· B_{21}	· B_{32}	· B_{13}
· B_{31}	· B_{12}	· B_{23}

$t = 3$

Thus, if B is mapped onto the torus in a "staggered start" fashion, we can arrange for the first row of processors to compute the first row of C.

If we stagger the second and third rows of A in a similar fashion, then we can arrange for all nine processors to perform a multiply-add at each step. In particular, if we set

A_{11} B_{11}	A_{12} B_{22}	A_{13} B_{33}
A_{22} B_{21}	A_{23} B_{32}	A_{21} B_{13}
A_{33} B_{31}	A_{31} B_{12}	A_{32} B_{23}

then with westward flow of the A_{ij} and northward flow of the B_{ij} we obtain

A_{12} B_{21}	A_{13} B_{32}	A_{11} B_{13}
A_{23} B_{31}	A_{21} B_{12}	A_{22} B_{23}
A_{31} B_{11}	A_{32} B_{22}	A_{33} B_{33}

$t = 1$

A_{13} B_{31}	A_{11} B_{12}	A_{12} B_{23}
A_{21} B_{11}	A_{22} B_{22}	A_{23} B_{33}
A_{32} B_{21}	A_{33} B_{32}	A_{31} B_{13}

$t = 2$

A_{11} B_{11}	A_{12} B_{22}	A_{13} B_{33}
A_{22} B_{21}	A_{23} B_{32}	A_{21} B_{13}
A_{33} B_{31}	A_{31} B_{12}	A_{32} B_{23}

$t = 3$

From this example we are ready to specify the general algorithm. We assume that at the start, Proc(i, j) houses A_{ij}, B_{ij}, and C_{ij}. To obtain the necessary staggering of the A data, we note that in processor row i the A_{ij} should be circulated westward $i - 1$ positions. Likewise, in the jth column of processors, the B_{ij} should be circulated northward $j - 1$ positions. This gives the following algorithm:

Algorithm 6.2.2 Suppose $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times n}$, and $C \in \mathbb{R}^{n\times n}$ are given and that $D = C + AB$. If each processor in a p_1-by-p_1 torus executes the following algorithm and $n = p_1 r$, then upon completion Proc(μ, λ) houses $D_{\mu\lambda}$ in local variable C_{loc}. Assume the following local memory initializations: p_1, (μ, λ) (the node id), *north*, *east*, *south*, and *west*, (the four neighbor id's), $row = 1 + (\mu - 1)r{:}\mu r$, $col = 1 + (\lambda - 1)r{:}\lambda r$, $A_{loc} = A(row, col)$, $B_{loc} = B(row, col)$, and $C_{loc} = C(row, col)$.

{Stagger the $A_{\mu j}$ and $B_{i\lambda}$. }
for $k = 1{:}\mu - 1$
  send$(A_{loc}, west)$; **recv**$(A_{loc}, east)$
end
for $k = 1{:}\lambda - 1$
  send$(B_{loc}, north)$; **recv**$(B_{loc}, south)$
end
for $k = 1{:}p_1$
  $C_{loc} = C_{loc} + A_{loc}B_{loc}$
  send$(A_{loc}, west)$
  send$(B_{loc}, north)$
  recv$(A_{loc}, east)$
  recv$(B_{loc}, south)$
end

```
{Unstagger the A_μj and B_iλ.}
for k = 1:μ − 1
      send(A_loc, east); recv(A_loc, west)
end
for k = 1:λ − 1
      send(B_loc, south); recv(B_loc, north)
end
```

It is not hard to show that the computation-to-communication ratio for this algorithm goes to zero as n/p_1 increases.

Problems

P6.2.1 Develop a ring implementation for Algorithm 6.2.1.

P6.2.2 An upper triangular matrix can be overwritten with its square without any additional workspace. Write a dynamically scheduled, shared-memory procedure for doing this.

Notes and References for Sec. 6.2

Matrix computations on 2-dimensional arrays are discussed in

H.T. Kung (1982). "Why Systolic Architectures?," *Computer 15*, 37-46.

D.P. O'Leary and G.W. Stewart (1985). "Data Flow Algorithms for Parallel Matrix Computations," *Comm. ACM 28*, 841-853.

B. Hendrickson and D. Womble (1994). "The Torus-Wrap Mapping for Dense Matrix Calculations on Massively Parallel Computers," *SIAM J. Sci. Comput. 15*, 1201-1226.

Various aspects of parallel matrix multiplication are discussed in

L.E. Cannon (1969). *A Cellular Computer to Implement the Kalman Filter Algorithm*, Ph.D. Thesis, Montana State University.

K.H. Cheng and S. Sahni (1987). "VLSI Systems for Band Matrix Multiplication," *Parallel Computing 4*, 239-258.

G. Fox, S.W. Otto, and A.J. Hey (1987). "Matrix Algorithms on a Hypercube I: Matrix Multiplication," *Parallel Computing 4*, 17-31.

J. Berntsen (1989). "Communication Efficient Matrix Multiplication on Hypercubes," *Parallel Computing 12*, 335-342.

H.J. Jagadish and T. Kailath (1989). "A Family of New Efficient Arrays for Matrix Multiplication," *IEEE Trans. Comput. 38*, 149-155.

P. Bjørstad, F. Manne, T.Sørevik, and M. Vajteršic (1992). "Efficient Matrix Multiplication on SIMD Computers," *SIAM J. Matrix Anal. Appl. 13*, 386-401.

K. Mathur and S.L. Johnsson (1994). "Multiplication of Matrices of Arbitrary Shape on a Data Parallel Computer," *Parallel Computing 20*, 919-952.

R. Mathias (1995). "The Instability of Parallel Prefix Matrix Multiplication," *SIAM J. Sci. Comp. 16*, 956-973.

6.3 Factorizations

In this section we present a pair of parallel Cholesky factorizations. To illustrate what a distributed memory factorization looks like, we implement the gaxpy Cholesky algorithm on a ring. A shared memory implementation of outer product Cholesky is also detailed.

6.3.1 A Ring Cholesky

Let us see how the Cholesky factorization procedure can be distributed on a ring of p processors. The starting point is the equation

$$G(\mu,\mu)G(\mu{:}n,\mu) \;=\; A(\mu{:}n,\mu) - \sum_{j=1}^{\mu-1} G(\mu,j)G(\mu{:}n,j) \;\equiv\; v(\mu{:}n)\,.$$

This equation is obtained by equating the μth column in the n-by-n equation $A = GG^T$. Once the vector $v(\mu{:}n)$ is found then $G(\mu{:}n,\mu)$ is a simple scaling:

$$G(\mu{:}n,\mu) \;=\; v(\mu{:}n)/\sqrt{v(\mu)}.$$

For clarity, we first assume that $n = p$ and that Proc(μ) initially houses $A(\mu{:}n,\mu)$. Upon completion, each processor overwrites its A-column with the corresponding G-column. For Proc(μ) this process involves $\mu - 1$ saxpy updates of the form

$$A(\mu{:}n,\mu) \longleftarrow A(\mu{:}n,\mu) - G(\mu,j)G(\mu{:}n,j)$$

followed by a square root and a scaling. The general structure of Proc(μ)'s node program is therefore as follows:

```
for j = 1:μ − 1
      Receive a G-column from the left neighbor.
      If necessary, send a copy of the received G-column to
            the right neighbor.
      Update A(μ:n, μ) .
end
Generate G(μ:n, μ) and, if necessary, send it to the
      right neighbor.
```

Thus Proc(1) immediately computes $G(1{:}n,1) = A(1{:}n,1)/\sqrt{A(1,1)}$ and sends it to Proc(2). As soon as Proc(2) receives this column it can generate $G(2{:}n,2)$ and pass it to Proc(3) etc.. With this pipelining arrangement we can assert that once a processor computes its G-column, it can quit. It also follows that each processor receives G-columns in ascending order, i.e., $G(1{:}n,1)$, $G(2{:}n,2)$, etc. Based on these observations we have

```
j = 1
while j < μ
      recv(g_loc(j:n), left)
      if μ < n
            send(g_loc(j:n), right)
      end
      A_loc(μ:n) = A_loc(μ:n) - g_loc(μ)g(μ:n)
      j = j + 1
end
A_loc(μ:n) = A_loc(μ:n)/sqrt(A_loc(μ))
if μ < n
      send(A_loc(μ:n), right)
end
```

Note that the number of received G-columns is given by $j-1$. If $j = \mu$, then it is time for Proc(μ) to generate and send $G(\mu{:}n, \mu)$.

We now extend this strategy to the general n case. There are two obvious ways to distribute the computation. We could require each processor to compute a contiguous set of G-columns. For example, if $n = 11$, $p = 3$, and $A = [\, a_1, \ldots, a_{11} \,]$, then we could distribute A as follows

$$\Big[\, \underbrace{a_1\ a_2\ a_3\ a_4}_{\text{Proc(1)}} \mid \underbrace{a_5\ a_6\ a_7\ a_8}_{\text{Proc(2)}} \mid \underbrace{a_9\ a_{10}\ a_{11}}_{\text{Proc(3)}} \,\Big] .$$

Each processor could then proceed to find the corresponding G columns. The trouble with this approach is that (for example) Proc(1) is idle after the fourth column of G is found even though much work remains.

Greater load balancing results if we distribute the computational tasks using the wrap mapping, i.e.,

$$\Big[\, \underbrace{a_1\ a_4\ a_7\ a_{10}}_{\text{Proc(1)}} \mid \underbrace{a_2\ a_5\ a_8\ a_{11}}_{\text{Proc(2)}} \mid \underbrace{a_3\ a_6\ a_9}_{\text{Proc(3)}} \,\Big] .$$

In this scheme Proc(μ) carries out the construction of $G(:, \mu{:}p{:}n)$. When a given processor finishes computing its G-columns, each of the other processors has at most one more G column to find. Thus if $n/p \gg 1$, then all of the processors are busy most of the time.

Let us examine the details of a wrap-distributed Cholesky procedure. Each processor maintains a pair of counters. The counter j is the index of the next G-column to be received by Proc(μ). A processor also needs to know the index of the next G-column that it is to produce. Note that if $col = \mu{:}p{:}n$, then Proc(μ) is responsible for $G(:, col)$ and that $L = \mathbf{length}(col)$ is the number of the G-columns that it must compute. We use q to indicate the status of G-column production. At any instant, $col(q)$ is the index of the next G-column to be produced.

Algorithm 6.3.1 Suppose $A \in \mathbb{R}^{n\times n}$ is symmetric and positive definite and that $A = GG^T$ is its Cholesky factorization. If each node in a p-processor ring executes the following program, then upon completion Proc(μ) houses $G(k{:}n, k)$ for $k = \mu{:}p{:}n$ in a local array $A_{loc}(1{:}n, L)$ where $L = \mathbf{length}(col)$ and $col = \mu{:}p{:}n$. In particular, $G(col(q){:}n, col(q))$ is housed in $A_{loc}(col(q){:}n, q)$ for $q = 1{:}L$. Assume the following local memory initializations: p, μ (the node id), *left* and *right* (the neighbor id's), n, and $A_{loc} = A(\mu{:}p{:}n, :)$.

```
j = 1; q = 1; col = μ:p:n; L = length(col)
while q ≤ L
      if j = col(q)
            { Form G(j:n, j) }
            A_loc(j:n, q) = A_loc(j:n, q)/sqrt(A_loc(j, q))
            if j < n
                  send(A_loc(j:n, q), right)
            end
            j = j + 1
            { Update local columns. }
            for k = q + 1:L
                  r = col(k)
                  A_loc(r:n, k) = A_loc(r:n, k) - A_loc(r, q)A_loc(r:n, q)
            end
            q = q + 1
      else
            recv(g_loc(j:n), left)
            Compute α, the id of the processor that generated the
                  received G-column.
            Compute β, the index of Proc(right)'s final column.
            if right ≠ α ∧ j < β
                  send(g_loc(j:n), right)
            end
            { Update local columns. }
            for k = q:L
                  r = col(k)
                  A.loc(r:n, k) = A_loc(r:n, k) - g_loc(r)g_loc(r:n)
            end
            j = j + 1
      end
end
```

To illustrate the logic of the pointer system we consider a sample 3-processor situation with $n = 10$. Assume that the three local values of q are 3,2, and

2 and that the corresponding values of $col(q)$ are 7, 5, and 6 :

$$[\underbrace{a_1\ a_4\ \overset{\downarrow}{a_7}\ a_{10}}_{\text{Proc(1)}} \mid \underbrace{a_2\ \overset{\downarrow}{a_5}\ a_8\ a_{11}}_{\text{Proc(2)}} \mid \underbrace{a_3\ \overset{\downarrow}{a_6}\ a_9}_{\text{Proc(3)}}]$$

Proc(2) now generates the fifth G-column and increment its q to 3.

The decision to pass a received G-column to the right neighbor needs to be explained. Two conditions must be fulfilled:

- The right neighbor must not be the processor which generated the G column. This way the circulation of the received G-column is properly terminated.
- The right neighbor must still have more G-columns to generate. Otherwise, a G-column will be sent to an inactive processor.

This kind of reasoning is quite typical in distributed memory matrix computations.

Let us examine the behavior of Algorithm 6.3.1 under the assumption that $n \gg p$. It is not hard to show that Proc(μ) performs

$$F(\mu) = \sum_{k=1}^{L} 2(n-(\mu+(k-1)p))(\mu+(k-1)p) \approx \frac{n^3}{3p}$$

flops. Each processor receives and sends just about every G-column. Using our communication overhead model (6.1.3), we see that the time each processor spends communicating is given by

$$m_\mu = \sum_{j=1}^{n} 2(\alpha_d + \beta_d(n-j)) \approx 2\alpha_d n + \beta_d n^2 .$$

If we assume that computation proceeds at R flops per second, then the computation/communication ratio for Algorithm 6.3.1 is approximately given by $(n/p)(1/3R\beta_d)$. Thus, communication overheads diminish in importance as n/p grows.

6.3.2 A Shared Memory Cholesky

Next we consider a shared memory implementation of the outer product Cholesky algorithm:

```
for k = 1:n
    A(k:n, k) = A(k:n, k)/sqrt(A(k, k))
    for j = k + 1:n
        A(j:n, j) = A(j:n, j) - A(j:n, k)A(j, k)
    end
end
```


The j-loop oversees an outer product update. The $n-k$ saxpy operations that make up its body are independent and easily parallelized. The scaling $A(k{:}n, k)$ can be carried out by a single processor with no threat to load balancing.

Algorithm 6.3.2 Suppose $A \in \mathbb{R}^{n\times n}$ is a symmetric positive definite matrix stored in a shared memory accessible to p processors. If each processor executes the following algorithm, then upon completion the lower triangular part of A is overwritten with its Cholesky factor. Assume the following initializations in each local memory: n, p and μ (the node id).

```
for k = 1:n
    if μ = 1
        v_loc(k:n) = A(k:n)
        v_loc(k:n) = v_loc(k:n)/sqrt(v_loc(k))
        A(k:n, k) = v_loc(k:n)
    end
    barrier
    v_loc(k + 1:n) = A(k + 1:n, k)
    for j = (k + μ):p:n
        w_loc(j:n) = A(j:n, j)
        w_loc(j:n) = w_loc(j:n) - v_loc(j)v_loc(j:n)
        A(j:n, j) = w_loc(j:n)
    end
    barrier
end
```

The scaling before the j-loop represents very little work compared to the outer product update and so it is reasonable to assign that portion of the computation to a single processor. Notice that two **barrier** statements are required. The first ensures that a processor does not begin working on the kth outer product update until the kth column of G is made available by Proc(1). The second **barrier** prevents the processing of the k+1st step to begin until the kth step is completely finished.

Problems

P6.3.1 It is possible to formulate a block version of Algorithm 6.3.1. Suppose $n = rN$. For $k = 1{:}N$ we (a) have Proc(1) generate $G(:, 1+(k-1)r{:}kr)$ and (b) have all processors participate in the rank r update of the trailing submatrix $A(kr+1{:}n, kr+1{:}n)$. See §4.2.6. The coarser granularity may improve performance if the individual processors like level-3 operations.

P6.3.2 Develop a shared memory QR factorization patterned after Algorithm 6.3.2. Proc(1) should generate the Householder vectors and all processors should share in the ensuing Householder update.

Notes and References for Sec. 6.3

General comments on distributed memory factorization procedures may be found in

G.A. Geist and M.T. Heath (1986). "Matrix Factorization on a Hypercube," in M.T. Heath (ed) (1986). *Proceedings of First SIAM Conference on Hypercube Multiprocessors*, SIAM Publications, Philadelphia, Pa.

I.C.F. Ipsen, Y. Saad, and M. Schultz (1986). "Dense Linear Systems on a Ring of Processors," *Lin. Alg. and Its Applic.* 77, 205–239.

D.P. O'Leary and G.W. Stewart (1986). "Assignment and Scheduling in Parallel Matrix Factorization," *Lin. Alg. and Its Applic.* 77, 275–300.

R.S. Schreiber (1988). "Block Algorithms for Parallel Machines," in *Numerical Algorithms for Modern Parallel Computer Architectures*, M.H. Schultz (ed), IMA Volumes in Mathematics and Its Applications, Number 13, Springer-Verlag, Berlin, 197–207.

S.L. Johnsson and W. Lichtenstein (1993). "Block Cyclic Dense Linear Algebra," *SIAM J. Sci.Comp. 14*, 1257–1286.

Papers specifically concerned with LU, Cholesky and QR include

R.N. Kapur and J.C. Browne (1984). "Techniques for Solving Block Tridiagonal Systems on Reconfigurable Array Computers," *SIAM J. Sci. and Stat. Comp. 5*, 701-719.

G.J. Davis (1986). "Column LU Pivoting on a Hypercube Multiprocessor," *SIAM J. Alg. and Disc. Methods* 7, 538–550.

J.M. Delosme and I.C.F. Ipsen (1986). "Parallel Solution of Symmetric Positive Definite Systems with Hyperbolic Rotations," *Lin. Alg. and Its Applic.* 77, 75–112.

A. Pothen, S. Jha, and U. Vemapulati (1987). "Orthogonal Factorization on a Distributed Memory Multiprocessor," in *Hypercube Multiprocessors*, ed. M.T. Heath, SIAM Press, 1987.

C.H. Bischof (1988). "QR Factorization Algorithms for Coarse Grain Distributed Systems," PhD Thesis, Dept. of Computer Science, Cornell University, Ithaca, NY.

G.A. Geist and C.H. Romine (1988). "LU Factorization Algorithms on Distributed Memory Multiprocessor Architectures," *SIAM J. Sci. and Stat. Comp. 9*, 639–649.

J.M. Ortega and C.H. Romine (1988). "The *ijk* Forms of Factorization Methods II: Parallel Systems," *Parallel Computing* 7, 149–162.

M. Marrakchi and Y. Robert (1989). "Optimal Algorithms for Gaussian Elimination on an MIMD Computer," *Parallel Computing 12*, 183–194.

Parallel triangular system solving is discussed in

R. Montoye and D. Laurie (1982). "A Practical Algorithm for the Solution of Triangular Systems on a Parallel Processing System," *IEEE Trans. Comp. C-31*, 1076–1082.

D.J. Evans and R. Dunbar (1983). "The Parallel Solution of Triangular Systems of Equations," *IEEE Trans. Comp. C-32*, 201–204.

C.H. Romine and J.M. Ortega (1988). "Parallel Solution of Triangular Systems of Equations," *Parallel Computing 6*, 109–114.

M.T. Heath and C.H. Romine (1988). "Parallel Solution of Triangular Systems on Distributed Memory Multiprocessors," *SIAM J. Sci. and Stat. Comp. 9*, 558–588.

G. Li and T. Coleman (1988). "A Parallel Triangular Solver for a Distributed-Memory Multiprocessor," *SIAM J. Sci. and Stat. Comp. 9*, 485–502.

S.C. Eisenstat, M.T. Heath, C.S. Henkel, and C.H. Romine (1988). "Modified Cyclic Algorithms for Solving Triangular Systems on Distributed Memory Multiprocessors," *SIAM J. Sci. and Stat. Comp. 9*, 589–600.

N.J. Higham (1995). "Stability of Parallel Triangular System Solvers," *SIAM J. Sci. Comp. 16*, 400–413.

Papers on the parallel computation of the LU and Cholesky factorization include

R.P. Brent and F.T. Luk (1982) "Computing the Cholesky Factorization Using a Systolic Architecture," *Proc. 6th Australian Computer Science Conf.* 295–302.

D.P. O'Leary and G.W. Stewart (1985). "Data Flow Algorithms for Parallel Matrix Computations," *Comm. of the ACM 28*, 841–853.

J.M. Delosme and I.C.F. Ipsen (1986). "Parallel Solution of Symmetric Positive Definite Systems with Hyperbolic Rotations," *Lin. Alg. and Its Applic.* 77, 75–112.

R.E. Funderlic and A. Geist (1986). "Torus Data Flow for Parallel Computation of Missized Matrix Problems," *Lin. Alg. and Its Applic.* 77, 149–164.

M. Costnard, M. Marrakchi, and Y. Robert (1988). "Parallel Gaussian Elimination on an MIMD Computer," *Parallel Computing 6*, 275–296.

Parallel methods for banded and sparse systems include

S.L. Johnsson (1985). "Solving Narrow Banded Systems on Ensemble Architectures," *ACM Trans. Math. Soft. 11*, 271–288.

S.L. Johnsson (1986). "Band Matrix System Solvers on Ensemble Architectures," in *Supercomputers: Algorithms, Architectures, and Scientific Computation*, eds. F.A. Matsen and T. Tajima, University of Texas Press, Austin TX., 196–216.

S.L. Johnsson (1987). "Solving Tridiagonal Systems on Ensemble Architectures," *SIAM J. Sci. and Stat. Comp. 8*, 354–392.

U. Meier (1985). "A Parallel Partition Method for Solving Banded Systems of Linear Equations," *Parallel Computers 2*, 33–43.

H. van der Vorst (1987). "Large Tridiagonal and Block Tridiagonal Linear Systems on Vector and Parallel Computers," *Parallel Comput. 5*, 45–54.

R. Bevilacqua, B. Codenotti, and F. Romani (1988). "Parallel Solution of Block Tridiagonal Linear Systems," *Lin.Alg. and Its Applic. 104*, 39–57.

E. Gallopoulos and Y. Saad (1989). "A Parallel Block Cyclic Reduction Algorithm for the Fast Solution of Elliptic Equations," *Parallel Computing 10*, 143–160.

J.M. Conroy (1989). "A Note on the Parallel Cholesky Factorization of Wide Banded Matrices," *Parallel Computing 10*, 239–246.

M. Hegland (1991). "On the Parallel Solution of Tridiagonal Systems by Wrap-Around Partitioning and Incomplete LU Factorization," *Numer. Math. 59*, 453–472.

M.T. Heath, E. Ng, and B.W. Peyton (1991). "Parallel Algorithms for Sparse Linear Systems," *SIAM Review 33*, 420–460.

V. Mehrmann (1993). "Divide and Conquer Methods for Block Tridiagonal Systems," *Parallel Computing 19*, 257–280.

P. Raghavan (1995). "Distributed Sparse Gaussian Elimination and Orthogonal Factorization," *SIAM J. Sci. Comp. 16*, 1462–1477.

Parallel QR factorization procedures are of interest in real-time signal processing. Details may be found in

W.M. Gentleman and H.T. Kung (1981). "Matrix Triangularization by Systolic Arrays," SPIE Proceedings, Vol. 298, 19–26.

D.E. Heller and I.C.F. Ipsen (1983). "Systolic Networks for Orthogonal Decompositions," *SIAM J. Sci. and Stat. Comp. 4*, 261–269.

M. Costnard, J.M. Muller, and Y. Robert (1986). "Parallel QR Decomposition of a Rectangular Matrix," *Numer. Math. 48*, 239–250.

L. Eldin and R. Schreiber (1986). "An Application of Systolic Arrays to Linear Discrete Ill-Posed Problems," *SIAM J. Sci. and Stat. Comp. 7*, 892–903.

F.T. Luk (1986). "A Rotation Method for Computing the QR Factorization," *SIAM J. Sci. and Stat. Comp. 7*, 452–459.

J.J. Modi and M.R.B. Clarke (1986). "An Alternative Givens Ordering," *Numer. Math. 43*, 83–90.

P. Amodio and L. Brugnano (1995). "The Parallel QR Factorization Algorithm for Tridiagonal Linear Systems," *Parallel Computing 21*, 1097–1110.

Parallel factorization methods for shared memory machines are discussed in

S. Chen, D. Kuck, and A. Sameh (1978). "Practical Parallel Band Triangular Systems Solvers," *ACM Trans. Math. Soft.* *4*, 270–277.
A. Sameh and D. Kuck (1978). "On Stable Parallel Linear System Solvers," *J. Assoc. Comp. Mach.* *25*, 81-91.
P. Swarztrauber (1979). "A Parallel Algorithm for Solving General Tridiagonal Equations," *Math. Comp.* *33*, 185-199.
S. Chen, J. Dongarra, and C. Hsuing (1984). "Multiprocessing Linear Algebra Algorithms on the Cray X-MP-2: Experiences with Small Granularity," *J. Parallel and Distributed Computing* *1*, 22–31.
J.J. Dongarra and A.H. Sameh (1984). "On Some Parallel Banded System Solvers," *Parallel Computing* *1*, 223–235.
J.J. Dongarra and R.E. Hiromoto (1984). "A Collection of Parallel Linear Equation Routines for the Denelcor HEP,' *Parallel Computing* *1*, 133–142.
J.J. Dongarra and T. Hewitt (1986). "Implementing Dense Linear Algebra Algorithms Using Multitasking on the Cray X-MP-4 (or Approaching the Gigaflop)," *SIAM J. Sci. and Stat. Comp.* *7*, 347–350.
J.J. Dongarra, A. Sameh, and D. Sorensen (1986). "Implementation of Some Concurrent Algorithms for Matrix Factorization," *Parallel Computing* *3*, 25–34.
A. George, M.T. Heath, and J. Liu (1986). "Parallel Cholesky Factorization on a Shared Memory Multiprocessor," *Lin. Alg. and Its Applic.* *77*, 165–187.
J.J. Dongarra and D.C. Sorensen (1987). "Linear Algebra on High Performance Computers," *Appl. Math. and Comp.* *20*, 57–88.
K. Dackland, E. Elmroth, and B. Kagstrom (1992). "Parallel Block Factorizations on the Shared Memory Multiprocessor IBM 3090 VF/600J," *International J. Supercomputer Applications*, *6*, 69–97.

Chapter 7

The Unsymmetric Eigenvalue Problem

§7.1 Properties and Decompositions
§7.2 Perturbation Theory
§7.3 Power Iterations
§7.4 The Hessenberg and Real Schur Forms
§7.5 The Practical QR Algorithm
§7.6 Invariant Subspace Computations
§7.7 The QZ Method for $Ax = \lambda Bx$

Having discussed linear equations and least squares, we now direct our attention to the third major problem area in matrix computations, the algebraic eigenvalue problem. The unsymmetric problem is considered in this chapter and the more agreeable symmetric case in the next.

Our first task is to present the decompositions of Schur and Jordan along with the basic properties of eigenvalues and invariant subspaces. The contrasting behavior of these two decompositions sets the stage for §7.2 in which we investigate how the eigenvalues and invariant subspaces of a matrix are affected by perturbation. Condition numbers are developed that permit estimation of the errors that can be expected to arise because of roundoff.

The key algorithm of the chapter is the justly famous QR algorithm. This procedure is the most complex algorithm presented in this book and its development is spread over three sections. We derive the basic QR iteration in §7.3 as a natural generalization of the simple power method. The next